Visita la web oficial del curso: **El Rincón del C** - Cursos, código fuente, DJGPP, Compiladores, artículos, noticias, ...

Contenido:

Bibliotecas (librerías) más comunes *(en preparación)*

Funciones matemáticas *(publicado el 23/9/99)*

Instalación de un compilador C: DJGPP *(revisado el 19/02/00)*

Autor

Contribuciones

Licencia

Fe de Erratas, nadie es perfecto

Encuesta

Otros Cursos de Gorka Urrutia

---

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Mostrando Información por pantalla

Contenido del capítulo

[Arriba]

Printf: Imprimir en pantalla

Siempre he creído que cuando empiezas con un nuevo lenguaje suele gustar el ver los resultados, ver que nuestro programa hace 'algo'. Por eso creo que el curso debe comenzar con la función printf, que sirve para sacar información por pantalla.

Para utilizar la función printf en nuestros programas debemos incluir la directiva:

```
#include <stdio.h>
```

al principio de programa. Como hemos visto en el programa *hola mundo*.

Si sólo queremos imprimir una cadena basta con hacer (no olvides el ";" al final):

```
printf( "Cadena" );
```

Esto resultará por pantalla:

```
Cadena
```

Lo que pongamos entre las comillas es lo que vamos a sacar por pantalla.

Si volvemos a usar otro printf, por ejemplo:

```
#include <stdio.h>

int main()
{
printf( "Cadena" );
printf( "Segunda" );

return 0;
}
```

Obtendremos:

```
CadenaSegunda
```

Este ejemplo nos muestra cómo funciona printf. Para escribir en la pantalla se usa un cursor que no vemos. Cuando escribimos algo el cursor va al final del texto. Cuando el texto llega al final de la fila, lo siguiente que pongamos irá a la fila siguiente. Si lo que queremos es sacar cada una en una línea deberemos usar "\n". Es el indicador de retorno de carro. Lo que hace es saltar el cursor de escritura a la línea siguiente:

```
#include <stdio.h>

int main()
{
printf( "Cadena\n" );
printf( "Segunda" );

return 0;
}
```

y tendremos:

```
Cadena
Segunda
```

También podemos poner más de una cadena dentro del printf:

```
printf( "Primera cadena" "Segunda cadena" );
```

Lo que no podemos hacer es meter *cosas* entre las cadenas:

```
printf( "Primera cadena" texto en medio  "Segunda cadena"
);
```

esto no es válido. Cuando el compilador intenta interpretar esta sentencia se encuentra *"Primera cadena"* y luego *texto en medio*, no sabe qué hacer con ello y da un error.

Pero ¿qué pasa si queremos imprimir el símbolo " en pantalla? Por ejemplo imaginemos que queremos escribir:

```
Esto es "extraño"
```

Si para ello hacemos:

```
printf( "Esto es "extraño"" );
```

obtendremos unos cuantos errores. El problema es que el símbolo " se usa para indicar al compilador el comienzo o el final de una cadena. Así que en realidad le estaríamos dando la cadena *"Esto es"*, luego *extraño* y luego otra cadena vacía *""*. Pues resulta que printf no admite esto y de nuevo tenemos errores.

La solución es usar \". Veamos:

```
printf( "Esto es \"extraño\"" );
```

Esta vez todo irá como la seda. Como vemos la contrabarra '\' sirve para indicarle al compilador que escriba caracteres que de otra forma no podríamos.

Esta contrabarra se usa en C para indicar al compilador que queremos meter símbolos especiales. Pero ¿Y si lo que queremos es usar '\' como un carácter normal y poner por ejemplo *Hola\Adiós*? Pues muy fácil, volvemos a usar '\':

```
printf( "Hola\\Adiós" );
```

y esta doble '\' indica a C que lo que queremos es mostrar una '\'.

Esto no ha sido mas que una introducción a printf. Luego volveremos sobre ella.

[Arriba]

Gotoxy: Posicionando el cursor (DOS)

Esta función sólo está disponible en compiladores de C que dispongan de la biblioteca <conio.h>

Hemos visto que cuando usamos *printf* se escribe en la posición actual del cursor y se mueve el cursor al final de la cadena que hemos escrito.

Vale, pero ¿qué pasa cuando queremos escribir en una posición determinada de la pantalla? La solución está en la función gotoxy. Supongamos que queremos escribir 'Hola' en la fila 10, columna 20 de la pantalla:

```
#include <stdio.h>
#include <conio.h>

int main()
{
gotoxy( 20, 10 );
printf( "Hola" );

return 0;
}
```

(Nota: para usar gotoxy hay que incluir la biblioteca *conio.h*).

Fíjate que primero se pone la columna (x) y luego la fila (y). La esquina superior izquierda es la posición (1, 1).

[Arriba]

Clrscr: Borrar la pantalla (DOS)

Ahora ya sólo nos falta saber cómo se borra la pantalla. Pues es tan fácil como usar:

```
clrscr()
```

(clear screen, borrar pantalla).

Esta función nó solo borra la pantalla, sino que además sitúa el cursor en la posición (1, 1), en la esquina superior izquierda.

```
#include <stdio.h>
#include <conio.h>

int main()
{
clrscr();
printf( "Hola" );

return 0;
}
```

Este método sólo vale para compiladores que incluyan el fichero stdio.h. Si tu sistema no lo tiene puedes consultar la sección siguiente.

[Arriba]

Borrar la pantalla (otros métodos)

Existen otras formas de borrar la pantalla aparte de usar stdio.h.

Si usas DOS:

```
system ("cls"); //Para DOS
```

Si usas Linux:

```
system ("clear"); // Para Linux
```

Otra forma válida para ambos sistemas:

```
char a[5]={27,'[','2','J',0}; /* Para ambos (en DOS
cargando antes ansi.sys) */
printf("%s",a);
```

[Arriba]

¿Qué sabemos hacer?

Bueno, ya hemos aprendido a sacar información por pantalla. Si quieres puedes practicar con las instrucciones printf, gotoxy y clrscr. Lo que hemos visto hasta ahora no tiene mucho secreto, pero ya veremos cómo la función printf tiene mayor complejidad.

[Arriba]

Ejercicios

Ejercicio 1: Busca los errores en el programa.

```
#include <stdio.h>

int main()
        {
        ClrScr();
        gotoxy( 10, 10 )
        printf( Estoy en la fila 10 columna 10 );

        return 0;
        }
```

Solución:

- ClrScr está mal escrito, debe ponerse todo en minúsculas, recordemos una vez más que el C diferencia las mayúsculas de las minúsculas. Además no hemos incluído la directiva *#include <conio.h>*, que necesitamos para usar clrscr() y gotoxy().
- Tampoco hemos puesto el punto y coma (;) después del gotoxy(10, 10). Después de cada instrucción debe ir un punto y coma.
- El último fallo es que el texto del printf no lo hemos puesto entre comillas. Lo correcto sería: printf("Estoy en la fila 10 columna 10");

Ejercicio 2. Escribe un programa que borre la pantalla y escriba en la primera línea su nombre y en la segunda su apellido:

Solución:

```
#include <stdio.h>
#include <conio.h>

int main()
```

```
        {
        clrscr();
        printf( "Gorka\n" );
        printf( "Urrutia" );

        return 0;
        }
```

También se podía haber hecho todo de golpe:

```
#include <stdio.h>
#include <conio.h>

int main()
        {
        clrscr();
        printf( "Gorka\nUrrutia" );

        return 0;
        }
```

Ejercicio 3. Escriba un programa que borre la pantalla y muestre el texto "estoy aqui" en la fila 10, columna 20 de la pantalla:

Solución:

```
#include <stdio.h>


#include <conio.h>




int main()


        {


        clrscr();


        gotoxy( 20, 10 );


        printf( "Estoy aqui" );



```

```
        return 0;


        }
```


cursoc@elrincondelc.com
http://www.elrincondelc.com/

Tipos de Datos

Contenido del Capítulo

[Arriba]

Introducción

Cuando usamos un programa es muy importante manejar datos. En C podemos almacenar los datos en variables. El contenido de las variables se puede ver o cambiar en cualquier momento. Estas variables pueden ser de distintos tipos dependiendo del tipo de dato que queramos meter. No es lo mismo guardar un nombre que un número. Hay que recordar también que la memoria del ordenador es limitada, así que cuando guardamos un dato, debemos usar sólo la memoria necesaria. Por ejemplo si queremos almacenar el número 400 usaremos una variable tipo *int* (la estudiamos más abajo) que ocupa sólo 16 bits, y no una de tipo *long* que ocupa 32 bits. Si tenemos un ordenador con 32Mb de Ram parece una tontería ponernos a ahorrar bits (1Mb=1024Kb, 1Kb=1024bytes, 1byte=8bits), pero si tenemos un programa que maneja muchos datos puede no ser una cantidad despreciable. Además ahorrar memoria es una buena costumbre.

(Por si alguno tiene dudas: No hay que confundir la memoria con el espacio en el disco duro. Son dos cosas distintas. La capacidad de ambos se mide en bytes, y la del disco duro suele ser mayor que la de la memoria Ram. La información en la Ram se pierde al apagar el ordenador, la del disco duro permanece. Cuando queremos guardar un fichero lo que necesitamos es espacio en el disco

duro. Cuando queremos ejecutar un programa lo que necesitamos es memoria Ram. La mayoría me imagino que ya lo sabeís, pero me he encontrado muchas veces con gente que los confunde.)

[Arriba]

Notas sobre los nombres de las variables

A las variables no se les puede dar cualquier nombre. No se pueden poner más que letras de la 'a' a la 'z' (la ñ no vale), números y el símbolo '_'. No se pueden poner signos de admiración, ni de interrogación... El nombre de una variable puede contener números, pero su primer carácter no puede serlo.

Ejemplos de nombres válidos:

```
camiones
numero
buffer
a1
j10hola29
num_alumnos
```

Ejemplos de nombres no válidos:

```
1abc
nombre?
num/alumnos
```

Tampoco valen como nombres de variable las **palabras reservadas** que usa el compilador. Por ejemplo: *for, main, do, while.*

Por último es interesante señalar que el C distingue entre mayúsculas y minúsculas. Por lo tanto:

```
Nombre
nombre
NOMBRE
```

serían tres variables distintas.

[Arriba]

El tipo Int

En una variable de este tipo se almacenan números enteros (sin decimales). El rango de valores que admite es -32767 a 32767. Cuando definimos una variable lo que estamos haciendo es decirle al compilador que nos reserve una zona de la memoria para almacenar datos de tipo int. Para guardarla necesitaremos 16 bits de la memoria del ordenador (2^{16}=32767). Para poder usar una variable primero hay que declararla (definirla). Hay que decirle al compilador que queremos crear una variable y hay que indicarle de qué tipo. Por ejemplo:

```
int numero;
```

Esto hace que declaremos una variable llamada *numero* que va a contener un número entero.

¿Pero dónde se declaran las variables?

Tenemos dos posibilidades, una es declararla como global y otra como local. Por ahora vamos a decir que global es aquella variable que se declara fuera de la función main y local la que se declara dentro:

Variable Global

```
#include <stdio.h>

int x;

int main()
        {
        }
```

Variable Local

```
#include <stdio.h>

int main()
        {
        int x;
        }
```

La diferencia práctica es que las variables globales se pueden usar en cualquier procedimiento. Las variables locales sólo pueden usarse en el procedimiento en el que se declaran. Como por ahora sólo tenemos el procedimiento (o función, o rutina, o subrutina, como prefieras) *main* esto no debe preocuparnos mucho por ahora. Cuando estudiemos cómo hacer un programa con más funciones aparte de main volveremos sobre el tema. Sin embargo debes saber que es buena costumbre usar variables locales que globales. Ya veremos por qué.

Podemos declarar más de una variable en una sola línea:

```
int x, y;
```

Mostrar variables por pantalla

Vamos a ir u poco más allá con la función printf. Supongamos que queremos mostrar el contenido de la variable

x
por pantalla:

```
printf( "%i", x );
```

Suponiendo que x valga 10 (x=10) en la pantalla tendríamos:

```
10
```

Empieza a complicarse un poco ¿no? Vamos poco a poco. ¿Recuerdas el símbolo "\" que usábamos para sacar ciertos caracteres? Bueno, pues el uso del "%" es parecido. "%i" no se muestra por pantalla, se sustituye por el valor de la variable que va detrás de las comillas. (%i, de integer=entero en inglés).

Para ver el contenido de dos variables, por ejemplo x e y, podemos hacer:

```
printf( "%i ", x );
printf( "%i", y );
```

resultado (suponiendo x=10, y=20):

```
10 20
```

Pero hay otra forma mejor:

```
printf( "%i %i", x, y );
```

... y así podemos poner el número de variables que queramos. Obtenemos el mismo resultado con menos trabajo. No olvidemos que por cada variable hay que poner un %i dentro de las comillas.

También podemos mezclar texto con enteros:

```
printf( "El valor de x es %i, ¡que bien!\n", x );
```

que quedará como:

```
El valor de x es 10, ¡que bien!
```

Como vemos %i al imprimir se sustituye por el valor de la variable.

Asignar valores a variables de tipo int

La asignación de valores es tan sencilla como:

```
x = 10;
```

También se puede dar un valor inicial a la variable cuando se define:

```
int x = 15;
```

También se pueden inicializar varias variables en una sola línea:

```
int x = 15, y = 20;
```

Hay que tener cuidado con lo siguiente:

```
int x, y = 20;
```

Podríamos pensar que x e y son igual a 20, pero no es así. La variable x está sin valor inicial y la variable 'y' tiene el valor 20.

Veamos un ejemplo para resumir todo:

```
#include <stdio.h>

int main()
{
int x = 10;

prinft( "El valor inicial de x es %i\n", x );
x = 50;
printf( "Ahora el valor es %i\n", x );
}
```

Cuya salida será:

```
El valor inicial de x es 10
Ahora el valor es 50
```

Importante! Si imprimimos una variable a la que no hemos dado ningún valor no obtendremos ningún error al compilar pero la variable tendrá un valor cualquiera. Prueba el ejemplo anterior quitando

```
int x = 10;
```

Puede que te imprima el valor 10 o puede que no.

[Arriba]

El tipo Char

Las variables de tipo char sirven para almacenar caracteres. Los caracteres se almacenan en realidad como números del 0 al 255. Los 128 primeros (0 a 127) son el ASCII estándar. El resto es el ASCII extendido y depende del idioma y del ordenador. Consulta **la tabla ASCII** en el anexo.

Para declarar una variable de tipo char hacemos:

```
char letra;
```

En una variable char **sólo podemos almacenar solo una letra**, no podemos almacenar ni frases ni palabras. Eso lo veremos más adelante (strings, cadenas). Para almacenar un dato en una variable char tenemos dos posibilidades:

```
letra = 'A';
o
letra = 65;
```

En ambos casos se almacena la letra 'A' en la variable. Esto es así porque el código ASCII de la letra 'A' es el 65.

Para imprimir un char usamos el símbolo %c (c de character=caracter en inglés):

```
letra = 'A';
printf( "La letra es: %c.", letra );
```

resultado:

```
La letra es A.
```

También podemos imprimir el valor ASCII de la variable usando %i en vez de %c:

```
letra = 'A';
printf( "El número ASCII de la letra %c es: %i.", letra,
letra );
```

resultado:

```
El código ASCII de la letra A es 65.
```

Como vemos la única diferencia para obtener uno u otro es el modificador (%c ó %i) que usemos.

Las variables tipo char se pueden usar (y de hecho se usan mucho) para almacenar enteros. Si necesitamos un número pequeño (entre -127 y 127) podemos usar una variable char (8bits) en vez de una int (16bits), con el consiguiente ahorro de memoria.

Todo lo demás dicho para los datos de tipo

`int`
se aplica también a los de tipo
`char`
.

Una curiosidad:

```
letra = 'A';
printf( "La letra es: %c y su valor ASCII es: %i\n",
letra, letra );
letra = letra + 1;
printf( "Ahora es: %c y su valor ASCII es: %i\n", letra,
letra );
```

En este ejemplo *letra* comienza con el valor 'A', que es el código ASCII 65. Al sumarle 1 pasa a tener el valor 66, que equivale a la letra 'B' (código ASCII 66). La salida de este ejemplo sería:

```
La letra es A y su valor ASCII es 65
Ahora es B y su valor ASCII es 66
```

[Arriba]

El modificador Unsigned

Este modificador (que significa *sin signo*) modifica el rango de valores que puede contener una variable. Sólo admite valores positivos. Si hacemos:

```
unsigned char variable;
```

Esta variable en vez de tener un rango de -128 a 128 pasa a tener un rango de 0 a 255.

[Arriba]

El tipo Float

En este tipo de variable podemos almacenar números decimales, no sólo enteros como en los anteriores. El rango de posibles valores es del 3,4E-38 al 3,4E38.

Declaración de una variable de tipo float:

```
float numero;
```

Para imprimir valores tipo float Usamos %f.

```
float num=4060.80;
printf( "El valor de num es : %f", num );

Resultado:

El valor de num es: 4060.80
```

Si queremos escribirlo en notación exponencial usamos %e:

```
float num = 4060.80;
printf( "El valor de num es: %e", num );

Que da como resultado:

El valor de num es: 4.06080e003
```

[Arriba]

El tipo Double

En las variables tipo double se almacenan números reales del 1,7E-307 al 1,7E308. Se declaran como *double*:

```
double numero;
```

Para imprimir se usan los mismos modificadores que en float.

[Arriba]

Cómo calcular el máximo valor que admite un tipo de datos

Lo primero que tenemos que conocer es el tamaño en bytes de ese tipo de dato. Vamos a ver un ejemplo con el tipo INT. Hagamos el siguiente programa:

```
#include <stdio.h>

int main()
     {
     int num1;

     printf( "El tipo int ocupa %i bytes\n", sizeof(int)
);
     }
```

Comprobado con DJGPP

En mi ordenador el resultado es:

```
El tipo int ocupa 4 bytes.
```

Como sabemos 1byte = 8bits. Por lo tanto el tipo int ocupa 4*8=32 bits.

Ahora para calcular el máximo número debemos elevar 2 al número de bits obtenido. En nuestro ejemplo: 2^32=4.294.967.296. Es decir en un int se podría almacenar un número entre 0 y 4.294.967.296.

Sin embargo esto sólo es cierto si usamos un tipo unsigned (sin signo, se hace añadiendo la palabra unsigned antes de int). Para los tipos normales tenemos que almacenar números positivos y negativos. Así que de los 4.294.967.296 posibles números la mitad serán positivos y la mitad negativos. Por lo tanto tenemos que dividir el número anterior entre 2 = 2.147.483.648. Como el 0 se considera positivo el rango de números posibles que se pueden almacenar en un int sería: -2.147.483.648 a 2.147.483.647.

Resumen:

1. Obtenemos el número de bytes.
2. Multiplicamos por ocho (ya lo tenemos en bits).
3. Elevamos 2 al número de bits.
4. Dividimos entre 2.

[Arriba]

Overflow: Qué pasa cuando nos saltamos el rango

El *overflow* es lo que se produce cuando intentamos almacenar en una variable un número mayor del máximo permitido. El comportamiento es distinto para variablesde números enteros y para variables de números en coma flotante.

Con números enteros

En mi ordenador y usando DJGPP bajo Dos el tamaño del tipo int es de 4bytes(4*8=32bits). El número máximo que se puede almacenar en una variable tipo int es por tanto 2.147.483.647 (ver apartado anterior). Si nos pasamos de este número el que se guardará será el siguiente pero empezando desde el otro extremo, es decir, el -2.147.483.648. El compilador seguramente nos dará un aviso (warning) de que nos hemos pasado.

```
#include <stdio.h>

int main()
     {
     int num1;
```

```
     num1 = 2147483648;
     printf( "El valor de num1 es: %i\n", num1 );
     }
```
Comprobado con DJGPP

El resultado que obtenemos es:

```
El valor de num1 es: -2147483648
```

Comprueba si quieres que con el número anterior (2.147.483.647) no pasa nada.

Con números en coma flotante

El comportamiento con números en coma flotante es distinto. Dependiendo del ordenador si nos pasamos del rango al ejecutar un programa se puede producir un error y detenerse la ejecución.

Con estos números también existe otro error que es el **underflow**. Este error se produce cuando almacenamos un número demasiado pequeño (3,4E-38 en float).

[Arriba]

Resumen de los tipos de datos en C

Esto es algo orientativo, depende del sistema.

Tipo	Datos almacenados	Nº de Bits	Valores posibles (Rango)	Rango usando unsigned
char	Caracteres	8	-128 a 128	0 a 255
int	enteros	16	-32.767 a 32.767	0 a 65.535
long	enteros largos	32	-2.147.483.647 a 2.147.483.647	0 a 4.294.967.295
float	Nums. reales (coma flotante)	32	3,4E-38 a 3,4E38	
double	Nums. reales (coma flotante doble)	64	1,7E-307 a 1,7E308	

Esto no siempre es cierto, depende del ordenador y del compilador. Para saber en nuestro caso qué tamaño tienen nuestros tipos de datos debemos hacer lo siguiente. Ejemplo para int:

```
#include <stdio.h>

int main()
        {
        printf( "Tamaño (en bits) de int = %i\n", sizeof(
int )*8 );

        return 0;
        }
```

Ya veremos más tarde lo que significa sizeof. Por ahora basta con saber que nos dice cual es el tamaño de una variable o un tipo de dato.

[Arriba]

Ejercicios

Ejercicio 1. Busque los errores:

```
#include <stdio.h>

int main()
        {
        int número;

        número = 2;

        return 0;
        }
```

Solución: Los nombres de variables no pueden llevar acentos, luego al compilar número dará error.

```
#include <stdio.h>

int main()
        {
        int numero;

        numero = 2;

        printf( "El valor es %i" Numero );

        return 0;
        }
```

Solución: Falta la coma después de "El valor es %i". Además la segunda vez **numero** está escrito con mayúsculas.

---

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Constantes (uso de #define)

Contenido del Capítulo

Introducción

Las constantes son aquellos datos que no pueden cambiar a lo largo de la ejecución de un programa.

```
#include <stdio.h>

int main()
        {
        int radio, perimetro;

        radio = 20;
        perimetro = 2 * 3.1416 * radio;


        printf( "El perímetro es: %i", perimetro  );

        return 0;
        }
```

radio y *perimetro* son variables, su valor puede cambiar a lo largo del programa. Sin embargo 20, 2 y 3.1416 son constantes, no hay manera de cambiarlas. El valor 3.1416 no cambia a lo largo del programa , ni entre ejecución y ejecución. Sólo cambiará cuando edites el programa y lo cambies tu mismo. Esto es obvio, y no tiene ningún misterio así que no le des vueltas. Sólo quería indicarlo porque en algunos libros le dan muchas vueltas.

[Arriba]

Constantes con nombre

Supongamos que tenemos que hacer un programa en el que haya que escribir unas cuantas veces 3.1416 (como todos sabreis es PI). Es muy fácil que nos confundamos alguna vez al teclearlo y al compilar el programa no tendremos ningún error, sin

embargo el programa no dará resultados correctos. Para evitar esto usamos las constantes con nombre. Al definir una constante con nombre estamos dando un nombre al valor, a 3.1416 le llamamos PI.

Estas constantes se definen de la manera siguiente:

```
#define nombre_de_la_constante valor_de_la_constante
```

Ejemplo:

```
#include <stdio.h>

#define PI     3.1416

int main()
        {
        int radio, perimetro;

        radio = 20;
        perimetro = 2 * PI * radio;

        printf( "El perímetro es: %i", perimetro  );

        return 0;
        }
```

De esta forma cada vez que el compilador encuentre el nombre PI lo sustituirá por 3.1416.

A una constante no se le puede dar un valor mientras se ejecuta, no se puede hacer PI = 20;. Sólo se le puede dar un valor con #define, y sólo una vez. Tampoco podemos usar el scanf para dar valores a una constante:

```
#define CONSTANTE      14

int main()
        {
        ...
        scanf( "%i", CONSTANTE );
        ...
        }
```

eso sería como hacer:

```
scanf( "%i", 14 );
```

Esto es muy grave, estamos diciendo que el valor que escribamos en scanf se almacene en la posición 14 de la memoria, lo que puede bloquear el ordenador.

Las constantes se suelen escribir en mayúsculas sólo se puede definir una constante por fila. **No se pone ";" al final.**

Las constantes nos proporcionan una mejor comprensión del código fuente. Mira el siguiente programa:

```
#include <stdio.h>

int main()
        {
        int precio;

        precio = ( 4 * 25 * 100 ) * ( 1.16 );

        printf( "El precio total es: %i", precio  );

        return 0;
        }
```

¿Quien entiende lo que quiere decir este programa? Es difícil si no imposible. Hagámoslo de otra forma.

```
#include <stdio.h>

#define CAJAS                   4
#define UNIDADES_POR_CAJA       25
#define PRECIO_POR_UNIDAD       100
#define IMPUESTOS               1.16

int main()
        {
        int precio;

        precio = ( CAJAS * UNIDADES_POR_CAJA * PRECIO_POR_UNIDAD ) * (
IMPUESTOS );

        printf( "El precio total es: %i", precio  );

        return 0;
        }
```

¿A que ahora se entiende mejor? Claro que sí. Los números no tienen mucho significado y si revisamos el programa un tiempo más tarde ya no nos acordaremos qué cosa era cada número. De la segunda forma nos enteraremos al momento.

También podemos definir una constante usando el valor de otras. Por supuesto las otras tienen que estar definidas antes:

```
#include <stdio.h>

#define CAJAS                   4
#define UNIDADES_POR_CAJA       25
#define PRECIO_POR_UNIDAD       100
#define PRECIO_POR_CAJA                 UNIDADES_POR_CAJA *
PRECIO_POR_UNIDAD
#define IMPUESTOS               1.16

int main()
        {
        int precio;

        precio = ( CAJAS * PRECIO_POR_CAJA ) * ( IMPUESTOS );

        printf( "El precio total es: %i", precio  );
```

```
return 0;
}
```

#define tiene más usos aparte de éste, ya los veremos en el capítulo de directivas.

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Manipulando datos (Operadores)

[Anterior] [Siguiente] [Contenido]

Contenido del capítulo

¿Qué es un operador?

Un operador sirve para manipular datos. Los hay de varios tipos: de asignación, de relación, lógicos, aritméticos y de manipulación de bits. En realidad los nombres tampoco importan mucho; aquí lo que queremos es aprender a programar, no aprender un montón de nombres.

[Arriba]

Operador de asignación

Este es un operador que ya hemos visto en el capítulo de Tipos de Datos. Sirve para dar un valor a una variable. Este valor puede ser un número que tecleamos directamente u otra variable:

```
a = 3;          /* Metemos un valor directamente */
```

```
o
a = b;          /* Le damos el valor de una variable */
```

Podemos dar valores a varias variables a la vez:

```
a = b = c = 10;         /* Damos a las variables a,b,c el
valor 10 */
```

También podemos asignar a varias variables el valor de otra de un sólo golpe:

```
a = b = c = d;          /* a,b,c toman el valor de d */
```

[Arriba]

Operadores aritméticos

Los operadores aritméticos son aquellos que sirven para realizar operaciones tales como suma, resta, división y multiplicación.

Si quieres saber cómo usar funciones matemáticas más complejas (exponentes, raíces, trigonométricas) vete al **capítulo correspondiente.**

[Arriba]

Operador (+) : Suma

Este operador permite sumar variables:

```
#include <stdio.h>

int main()
        {
        int a = 2;
        int b = 3;
        int c;

        c = a + b;
        printf ( "Resultado = %i\n", c );
        }
```

El resultado será 5 obviamente.

Por supuesto se pueden sumar varias variables o variables más constantes:

```
#include <stdio.h>

int main()
        {
        int a = 2;
        int b = 3;
        int c = 1;
        int d;

        d = a + b + c + 4;
        printf ( "Resultado = %i\n", c );
        }
```

El resultado es 10.

Podemos utilizar este operador para incrementar el valor de una variable:

```
x = x + 5;
```

Pero existe una forma abreviada:

```
x += 5;
```

Esto suma el valor 5 al valor que tenía la variable x. Veamos un ejemplo:

```
#include <stdio.h>

int main()
        {
        int x, y;

        x = 3;
        y = 5;

        x += 2;
        printf( "x = %i\n", x );
        x += y;         /* esto equivale a x = x + y */
        printf( "x = %i\n", x );
        }
```

Resultado:

```
x = 5
x = 10
```

[Arriba]

Operador (++) : Incremento

Este operador equivale a sumar uno a la variable:

```
#include <stdio.h>

int main()
        {
        int x = 5;

        printf ( "Valor de x = %i\n", x );
        x++;
        printf ( "Valor de x = %i\n", x );
        }
```

Resultado:

```
Valor de x = 5
Valor de x = 6
```

Se puede poner antes o después de la variable.

[Arriba]

Operador (-) : Resta/Negativo

Este operador tiene dos usos, uno es la resta que funciona como el operador suma y el otro es cambiar de signo.

Resta:

```
x = x - 5;
```

Para la operación resta se aplica todo lo dicho para la suma. Se puede usar también como: **x -= 5;**.

Pero también tiene el uso de cambiar de signo. Poniendolo delante de una variable o constante equivale a multiplicarla por -1.

```
#include <stdio.h>

int main()
        {
        int a, b;

        a = 1;

        b = -a;
        printf( "a = %i, b = %i\n", a, b );
        }
```

Resultado: a = 1, b = -1. No tiene mucho misterio.

[Arriba]

Operador (--) : Decremento

Es equivalente a ++ pero en vez de incrementar disminuye el valor de la variable. Equivale a restar uno a la variable.

[Arriba]

Operador (*) : Multiplicación y punteros

Este operador sirve para multiplicar y funciona de manera parecida a los anteriores.

También sirve para definir y utilizar punteros, pero eso lo veremos más tarde.

[Arriba]

Operador (/) : División

Este funciona también como los anteriores pero hay que tener cuidado. Si dividimos dos número en coma flotante (tipo float) tenemos las división con sus correspondientes decimales. Pero si dividimos dos enteros obtenemos un número entero. Es decir que si dividimos 4/3 tenemos como resultado 1. El redondeo se hace por truncamiento, simplemente se eliminan los decimales y se deja el entero.

Si dividimos dos enteros el resultado es un número entero, aunque luego lo saquemos por pantalla usando %f o %d no obtendremos la parte decimal.

Cuando dividimos dos enteros, si queremos saber cual es el resto (o módulo) usamos el operador %, que vemos más abajo.

[Arriba]

Operador (%) : Resto

Si con el anterior operador obteníamos el módulo o cociente de una división entera con éste podemos tener el resto. No funciona más que con enteros, no vale para números float o double.

Cómo se usa:

```
#include <stdio.h>

int main()
        {
        int a, b;

        a = 18;
        b = 5;
        printf( "Resto de la división: %d \n", a % b );
        }
```

[Arriba]

Operadores de comparación

Los operadores de condición se utilizan para comprobar las condiciones de las sentencias de control de flujo (las estudiaremos en el capítulo sentencias).

Cuando se evalúa una condición el resultado que se obtiene es 0 si no se cumple y un número distinto de 0 si se cumple. Normalmente cuando se cumplen devuelven un 1.

Los operadores de comparación son:

==	igual que	se cumple si son iguales
!=	distinto que	se cumple 1 si son diferentes
>	mayor que	se cumple si el primero es mayor que el segundo
<	menor que	se cumple si el primero es menor que el segundo
>=	mayor o igual que	se cumple si el primero es mayor o igual que el segundo
<=	menor o igual que	se cumple si el primero es menor o igual que el segundo

Veremos la aplicación de estos operadores en el capítulo **Sentencias**. Pero ahora vamos a ver unos ejemplos:

```
#include <stdio.h>

int main()
        {
        printf( "10 > 5 da como resultado %i\n", 10>5 );
        printf( "10 > 5 da como resultado %i\n", 10>5 );
        printf( "5== 5 da como resultado %i\n", 5==5 );
        printf( "10==5 da como resultado %i\n", 10==5 );
        }
```

No sólo se pueden comparar constantes, también se pueden comparar variables.

[Arriba]

Operadores lógicos

Estos son los que nos permiten unir varias comparaciones: 10>5 **y** 6==6. Los operadores lógicos son: AND (&&), OR (||), NOT(!).

Operador && (AND, en castellano Y): Devuelve un 1 si se cumplen dos condiciones.

```
printf( "Resultado: %i", (10==10 && 5>2 );
```

Operador || (OR, en castellano O): Devuelve un 1 si se cumple una de las dos condiciones.

Operador ! (NOT, negación): Si la condición se cumple NOT hace que no se cumpla y viceversa.

Ver el capítulo **Sentencias**, sección **Notas sobre las condiciones** para más información.

[Arriba]

Introducción a los bits y bytes

En esta sección voy a describir lo que son los bytes y los bits. No voy a descubrir nada nuevo, así que la mayoría se podrán saltar esta sección.

Supongo que todo el mundo sabe lo que son los bytes y los bits, pero por si acaso allá va. Los bits son la unidad de información más pequeña, digamos que son la base para almacenar la información. Son como los átomos a las moléculas. Los valores que puede tomar un bit son 0 ó 1. Si juntamos ocho bits tenemos un byte.

Un byte puede tomar 256 valores diferentes (de 0 a 255). ¿Cómo se consigue esto? Imaginemos nuestro flamante byte con sus ocho bits. Supongamos que los ocho bits valen cero. Ya tenemos el valor 0 en el byte. Ahora vamos a darle al último byte el valor 1.

```
00000001 -> 1
```

Este es el uno para el byte. Ahora vamos a por el dos y el tres:

```
00000010 -> 2
00000011 -> 3
```

y así hasta 255. Como vemos con ocho bits podemos tener 256 valores diferentes, que en byte corresponden a los valores entre 0 y 255.

[Arriba]

Operadores de bits

Ya hemos visto que un byte son ocho bits. Pues bien, con los operadores de bits podemos manipular las variables *por dentro*. Los diferentes operadores de bits son:

Operador	Descripción
|	OR (O)
&	AND (Y)
^	XOR (O exclusivo)
~	Complemento a uno o negación
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

Operador | (OR):

Toma los valores y hace con ellos la operación OR. Vamos a ver un ejemplo:

```
#include <stdio.h>

int main()
        {
        printf( "El resultado de la operación 235 | 143
es: %i\n", 235 | 143 );
        }
```

Se obtiene:

```
El resultado de la operación 235 | 143 es: 239
```

Veamos la operación a nivel de bits:

```
235 -> 11101011
143 -> 10001111 |
239 -> 11101111
```

La operación OR funciona de la siguiente manera: Tomamos los bits dos a dos y los comparamos si alguno de ellos es uno, se obtiene un uno. Si ambos son cero el resultado es cero. Primero se compara los dos primeros (el primero de cada uno de los números, 1 y 1 -> 1), luego la segunda pareja (1 y 0 -> 1) y así sucesivamente.

Operador & (AND):

Este operador compara los bits también dos a dos. Si ambos son 1 el resultado es 1. Si no, el resultado es cero.

```
#include <stdio.h>

int main()
        {
        printf( "El resultado de la operación 170 & 155
es: %i\n", 170 & 155 );
        }
```

Tenemos:

```
El resultado de la operación 170 & 155 es: 138
```

A nivel de bits:

```
170 -> 10101010
155 -> 10011011 &
138 -> 10001010
```

Operador ^ (XOR):

Compara los bits y los pone a unos si son **distintos**.

```
235 -> 11101011
143 -> 10001111 ^
100 -> 01100100
```

Operador ~ (Complemento a uno):

Este operador acepta un sólo dato (operando) y pone a 0 los 1 y a 1 los 0, es decir los invierte. Se pone delante del operando.

```
#include <stdio.h>

int main()
        {
        printf( "El resultado de la operación ~152 es:
%i\n", ~152 );
        }

El resultado de la operación ~152 es: 103
```

```
152 -> 10011000 ~
103 -> 01100111
```

Operador >> (Desplazamiento a la derecha):

Este operador mueve cada bit a la derecha. El bit de la izquierda se pone a cero, el de la derecha se pierde. Si realizamos la operación inversa no recuperamos el número original. El formato es:

```
variable o dato >> número de posiciones a desplazar
```

El *número de posiciones a desplazar* indica cuantas veces hay que mover los bits hacia la derecha. Ejemplo:

```
#include <stdio.h>

int main()
        {
        printf( "El resultado de la operación 150 >> 2
es: %i\n", ~152 );
        }

El resultado de la operación 150 >> 2 es: 37
```

Veamos la operación paso a paso. Esta operación equivale a hacer dos desplazamientos a la derecha:

```
150 -> 10010110  Número original
 75 -> 01001011 Primer desplazamiento. Entra un cero por
la izquierda,
                el cero de la derecha se pierde y los
demás se mueven a la derecha.
 37 -> 00100101 Segundo desplazamiento.
```

NOTA: Un desplazamiento a la izquierda equivale a dividir por dos. Esto es muy interesante porque el desplazamiento es más rápido que la división. Si queremos optimizar un programa esta es una buena idea. Sólo sirve para dividir entre dos. Si hacemos dos desplazamientos sería dividir por dos dos veces, no por tres.

Operador << (Desplazamiento a la izquierda):

Funciona igual que la anterior pero los bits se desplazan a la izquierda. Esta operación equivale a multiplicar por 2.

[Arriba]

Operador Sizeof

Este es un operador muy útil. Nos permite conocer el tamaño **en bytes** de una variable. De esta manera no tenemos que preocuparnos en recordar o calcular cuanto ocupa. Además el tamaño de una variable cambia de un compilador a otro, es la mejor forma de asegurarse. Se usa poniendo el nombre de la variable después de sizeof y separado de un espacio:

```
#include <stdio.h>

int main()
        {
        int variable;

        printf( "Tamaño de la variable: %i\n", sizeof
variable );
        }
```

Tambíen se puede usar con los especificadores de tipos de datos (char, int, float, double...). Pero en éstos se usa de manera diferente, hay que poner el especificador entre paréntesis:

```
#include <stdio.h>

int main()
        {
        printf( "Las variables tipo int ocupan: %i\n",
sizeof(int) );
        }
```

[Arriba]

Otros operadores

Existen además de los que hemos visto otros operadores. Sin embargo ya veremos en sucesivos capítulos lo que significa cada uno.

[Arriba]

Orden de evaluación de Operadores

Debemos tener cuidado al usar operadores pues a veces podemos tener resultados no esperados si no tenemos en cuenta su orden de evaluación. Vamos a ver la lista de precedencias, cuanto más arriba se evalúa antes:

Precedencia
() [] -> .
! ~ ++ -- (molde) * & sizeof (El * es el de puntero)
* / % (El * de aquí es el de multiplicación)
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /=
,

Por ejemplo imaginemos que tenemos la siguiente operación:

```
10 * 2 + 5
```

Si vamos a la tabla de precedencias vemos que el * tiene un orden superior al +, por lo tanto primero se hace el producto 10*2=20 y luego la suma 20+5=25. Veamos otra:

```
10 * ( 2 + 5 )
```

Ahora con el paréntesis cambia el orden de evaluación. El que tiene mayor precedencia ahora es el paréntesis, se ejecuta primero. Como dentro del

paréntesis sólo hay una suma se evalúa sin más, 2+5=7. Ya solo queda la multiplicación 10*7=70. Otro caso:

```
10 * ( 5 * 2 +3 )
```

Como antes, el que mayor precedencia tiene es el paréntesis, se evalúa primero. Dentro del paréntesis tenemos producto y suma. Como sabemos ya se evalúa primero el producto, 5*2=10. Seguimos en el paréntesis, nos queda la suma 10+3=13. Hemos acabado con el paréntesis, ahora al resto de la expresión. Cogemos la multiplicación que queda: 10*13=130.

Otro detalle que debemos cuidar son los operadores ++ y --. Estos tienen mayor precedencia que las demás operaciones aritméticas (+, -, *, /, %). Por ejemplo:

```
10 * 5 ++
```

Puede parecer que primero se ejecutará la multiplicación y luego el ++. Pero si vamos a la tabla de precedencias vemos que el ++ está por encima de * (de multiplicación), por lo tanto se evaluará primero 5++=6. 10*6=60.

Es mejor no usar los operadores ++ y -- mezclados con otros, pues a veces obtenemos resultados inesperados. Por ejemplo:

```
#include <stdio.h>

int main()
        {
        int a, b;

        a = 5;
        b = a++;
        printf( "a = %i, b = %i\n", a, b );
        }
```

Este ejemplo en unos compiladores dará a = 6, b = 5 y en otros a = 6 y b = 6. Por favor si podeis, escribidme diciendo qué valores os salen a vosotros y qué compilador usais.

Para asegurarse lo mejor sería separar la línea donde se usa el ++ y el =:

```
#include <stdio.h>

int main()
```

```
        {
        int a, b;

        a = 5;
        a++;
        b = a;
        printf( "a = %i, b = %i\n", a, b );
        }
```

[Arriba]

Ejercicios

Ejercicio 1: En este programa hay un fallo muy gordo y muy habitual en programación. A ver si lo encuentras:

```
#include <stdio.h>

int main()
        {
        int a, c;

        a = 5;
        c += a +5;
        }
```

Solución:

Cuando calculamos el valor de 'c' sumamos a+5 (=10) al valor de 'c'. Pero resulta que 'c' no tenía ningún valor indicado por nosotros. Estamos usando la variable 'c' sin haberle dado valor. En algunos compiladores el resultado será inesperado. Este es un fallo bastante habitual, usar variables a las que no hemos dado ningún valor.

Ejercicio 2: ¿Cual será el resultado del siguiente programa?

```
#include <conio.h>
#include <stdio.h>

int main()
     {
     int a, b, c;

     a = 5;
     b = ++a;
     c = ( a + 5 * 2 ) * ( b + 6 / 2 ) + ( a * 2 );
```

```
     printf( "%i, %i, %i", a, b, c );
     }
```

Solución:

El resultado es 156. En la primera a vale 5. Pero en la segunda se ejecuta b = ++a = ++5 = 6. Tenemos a = b = 6.

Ejercicio 3: Escribir un programa que compruebe si un número es par o impar.

Solución:

```c
#include <stdio.h>

int main() {
    char palabra[100];
    int a;

    a = 124;
    if ( a % 2 == 0 )
        printf( "%d es par\n", a );
    else
        printf( "%d es impar\n", a );
    printf( "\n" );
    system( "pause" );
    return 0;
}
```

Fichero: cap5_ejercicio3.c

Para comprobar si un número es par o impar podemos usar el operador '%'. Si al calcular el resto de dividir un número por 2 el resultado es cero eso indica que el número es par. Si el resto es distinto de cero el número es impar.

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Introducir datos por teclado

Contenido del capítulo

Introducción

Algo muy usual en un programa es esperar que el usuario introduzca datos por el teclado. Para ello contamos con varias posibilidades: Usar las funciones de la biblioteca estándar, crear nuestras propias interrupciones de teclado (MS-Dos) o usar funciones de alguna biblioteca diferente (como por ejemplo Allegro).

Nosotros en este capítulo vamos a estudiar la primera, usando las funciones de la biblioteca estándar. Pero veamos por encima las otras posibilidades.

Las funciones estándar están bien para un programa sencillito. Pero cuando queremos hacer juegos por ejemplo, no suelen ser suficiente. Demasiado lentas o no nos dan todas las posibilidades que buscamos, como comprobar si hay varias teclas pulsadas. Para solucionar esto tenemos dos posibilidades:

La más coplicada es crear nuestras propias interrupciones de teclado. ¿Qué es una interrupción de teclado? Es un pequeño programa en memoria que se ejecuta contínuamente y comprueba el estado del teclado. Podemos crear uno nuestro y hacer que el ordenador use el que hemos creado en vez del suyo. Este tema no lo vamos a tratar ahora, quizás en algún capítulo posteror.

Otra posibilidad más sencilla es usar una biblioteca que tenga funciones para controlar el teclado. Por ejemplo si usamos la biblioteca Allegro, ella misma hace todo el trabajo y nosotros no tenemos más que recoger sus frutos con un par de sencillas instrucciones. Esto soluciona mucho el

trabajo y nos libra de tener que aprender cómo funcionan los aspectos más oscuros del control del teclado.

Vamos ahora con las funciones de la biblioteca estándar.

[Arriba]

Scanf

El uso de scanf es muy similar al de printf con una diferencia, nos da la posibilidad de que el usuario introduzca datos en vez de mostrarlos. No nos permite mostrar texto en la pantalla, por eso si queremos mostrar un mensaje usamos un printf delante. Un ejemplo:

```
#include <stdio.h>

int main()
    {
    int num;
    printf( "Introduce un número " );
    scanf( "%i", &num );
    printf( "Has tecleado el número %i\n", num );
    return 0;
    }
```

Primero vamos a ver unas nota de estética, para hacer los programas un poco más elegantes. Parece una tontería, pero los pequeños detalles hacen que un programa gane mucho. El scanf no mueve el cursor de su posición actual, así que en nuestro ejemplo queda:

```
    Introduce un número _  /* La barra horizontal
indica dónde esta el cursor */
```

Esto es porque en el printf no hemos puesto al final el símbolo de salto de línea '\n'. Además hemos dejado un espacio al final de *Introduce un número:* para que así cuando tecleemos el número no salga pegado al mensaje. Si no hubiésemos dejado el espacio quedaría así al introducir el número 120 (es un ejemplo):

```
    Introduce un número120
```

Bueno, esto es muy interesante pero vamos a dejarlo y vamos al grano. Veamos cómo funciona el scanf. Lo primero nos fijamos que hay una cadena entre comillas. Esta es similar a la de printf, nos sirve para indicarle al compilador qué tipo de datos estamos pidiendo. Como en este caso es un integer usamos %i. Después de la coma tenemos la variable donde almacenamos el dato, en este caso 'num'.

Fíjate que en el scanf la variable 'num' lleva delante el símbolo **&**, este es muy importante, sirve para indicar al compilador cual es la dirección (o posición en la memoria) de la variable. Por ahora no te preocupes por eso, ya volveremos más adelante sobre el tema.

Podemos preguntar por más de una variable a la vez en un sólo scanf, hay que poner un %i por cada variable:

```
#include <stdio.h>

int main()
    {
    int a, b, c;

    printf( "Introduce tres números: " );
    scanf( "%i %i %i", &a, &b, &c );
    printf( "Has tecleado los números %i %i %i\n",
a, b, c );
    return 0;
    }
```

De esta forma cuando el usuario ejecuta el programa debe introducir los tres datos separados por un espacio.

También podemos pedir en un mismo scanf variables de distinto tipo:

```
#include <stdio.h>

int main()
    {
    int a;
    float b;

    printf( "Introduce dos números: " );
    scanf( "%i %f", &a, &b );

    printf( "Has tecleado los números %i %f\n", a,
b );
    return 0;
    }
```

A cada modificador (%i, %f) le debe corresponder una variable de su mismo tipo. Es decir, al poner un %i el compilador espera que su

variable correspondiente sea de tipo int. Si ponemos %f espera una variable tipo float.

[Arriba]

Getch y getche

Si lo que queremos es que el usuario introduzca un caracter por el teclado usamos las funciones getch y getche. Estas esperan a que el usuario introduzca un carácter por el teclado. La diferencia entre getche y getch es que la primera saca por pantalla la tecla que hemos pulsado y la segunda no (la e del final se refiere a *echo*=eco). Ejemplos:

```
#include <stdio.h>

int main()
    {
    char letra;

    printf( "Introduce una letra: " );
    fflush( stdout );
    letra = getche();
    printf( "\nHas introducido la letra: %c",
letra );
    return 0;
    }
```

Resultado:

```
Introduce una letra: a
Has introducido la letra: a
```

Quizás te estés preguntando qué es eso de fflush(stdout). Pues bien, cuando usamos la función printf, no escribimos directamente en la pantalla, sino en una memoria intermedia (lo que llaman un bufer). Cuando este bufer se llena o cuando metemos un carácter '\n' es cuando se envia el texto a la pantalla. En este ejemplo yo quería que apareciera el mensaje *Introduce una letra:* y el cursor se quedara justo después, es decir, sin usar '\n'. Si se hace esto, en algunos compiladores el mensaje no se muestra en pantalla hasta que se pulsa una tecla (pruébalo en el tuyo). Y la función fflush(stdout) lo que hace es enviar a la pantalla lo que hay en ese bufer.

Y ahora un ejemplo con getch:

```
#include <stdio.h>

int main()
    {
    char letra;

    printf( "Introduce una letra: " );
    fflush( stdout );
    letra = getch();
    printf("\n has introducido la letra :%c",
letra );
    return 0;
    }
```

Resultado:

```
Introduce una letra:
Has introducido la letra: a
```

Como vemos la única diferencia es que en el primer ejemplo se muestra en pantalla lo que escribimos y en el segundo no.

[Arriba]

Ejercicios

Ejercicio 1: Busca el error en el siguiente programa:

```
#include <stdio.h>



int main() {

    int numero;



    printf( "Introduce un número: " );

    scanf( "%d", numero );
```

```
    printf( "\nHas introducido el número %d.\n",
numero );


    return 0;


}
```

Solución:

A la variable número le falta el '&' con lo que no estamos indicando al programa la dirección de la variable y no obtendremos el resultado deseado. Haz la prueba y verás que el mensaje "Has introducido el número X" no muestra el número que habías introducido.

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Sentencias de control de flujo

[Anterior] [Siguiente] [Contenido]

Contenido del Capítulo

Introducción

Hasta ahora los programas que hemos visto eran lineales. Comenzaban por la primera instrucción y acababan por la última, ejecutándose todas una sola vez. . Lógico ¿no?. Pero resulta que muchas veces no es esto lo que queremos que ocurra. Lo que nos suele interesar es que dependiendo de los valores de los datos se ejecuten unas instrucciones y no otras. O también puede que queramos repetir unas instrucciones un número determinado de veces. Para esto están las sentencia de control de flujo.

[Arriba]

Bucles

Los bucles nos ofrecen la solución cuando queremos repetir una tarea un número determinado de veces. Supongamos que queremos escribir 100 veces la palabra *hola*. Con lo que sabemos hasta ahora haríamos:

```
    #include <stdio.h>

    int main()
            {
            printf( "Hola\n");
            printf( "Hola\n");
            printf( "Hola\n");
            printf( "Hola\n");
            printf( "Hola\n");
            ... (y así hasta 100 veces)
            }
```

¡Menuda locura! Y si queremos repetirlo más veces nos quedaría un programa de lo más largo.

Sin embargo usando un bucle *for* el programa quedaría:

```
    #include <stdio.h>
    int main()
            {
            int i;
            for ( i=0 ; i<100 ; i++ )
                    {
                    printf( "Hola\n" );
                    }
            }
```

Con lo que tenemos un programa más corto.

[Arriba]

El bucle For

El formato del bucle for es el siguiente:

```
    for( dar valores iniciales ; condiciones ;
incrementos )
            {
            conjunto de intrucciones a ejecutar en
el bucle
            }
```

Vamos a verlo con el ejemplo anterior:

```
    ...
    for ( i=0 ; i<100 ; i++ )
    ...
```

En este caso asignamos un valor inicial a la variable *i*. Ese valor es *cero*. Esa es la parte de *dar valores iniciales*. Luego tenemos *i<100*. Esa es la parte *condiciones*. En ella ponemos la condición es que i sea menor que 100, de modo que el bucle se ejecutará mientras i sea menor que 100. Es decir, mientras se cumpla la condición. Luego tenemos la parte de *incrementos*, donde indicamos cuánto se incrementa la variable.

Como vemos, el for va delante del grupo de instrucciones a ejecutar, de manera que si la condición es falsa, esas instrucciones no se ejecutan ni una sola vez.

Cuidado: No se debe poner un ";" justo después de la sentencia for, pues entonces sería un bucle vacío y las instrucciones siguientes sólo se ejecutarían una vez. Veamoslo con un ejemplo:

```
    #include <stdio.h>
    int main()
            {
            int i;
            for ( i=0 ; i<100 ; i++ );
    /* Cuidado con este punto y coma */
                    {
                    printf( "Hola\n" );
                    }
            }
```

Este programa sólo escribirá en pantalla

```
Hola
```

una sola vez.

Tambíen puede suceder que quieras ejecutar un cierto número de veces una sóla instrucción (como sucede en nuestro ejemplo). Entonces no necesitas las llaves "{}":

```
    #include <stdio.h>
    int main()
            {
            int i;
```

```
            for ( i=0 ; i<100 ; i++ ) printf(
"Hola\n" );
            }
```

o también:

```
            for ( i=0 ; i<100 ; i++ )
                    printf( "Hola\n" );
```

Sin embargo, yo me he encontrado muchas veces que es mejor poner las llaves aunque sólo haya una instrucción; a veces al añadir una segunda instrucción más tarde se te olvidan las comillas y no te das cuenta. Parece una tontería, pero muchas veces, cuando programas, son estos los pequeños fallos los que te vuelven loco.

En otros lenguajes, como Basic, la sentencia for es muy rígida. En cambio en C es muy flexible. Se puede omitir cualquiera de las secciones (inicialización, condiciones o incrementos). También se pueden poner más de una variable a inicializar, más de una condición y más de un incremento. Por ejemplo, el siguiente programa sería perfectamente correcto:

```
#include <stdio.h>

int main()
    {
    int i, j;

    for( i=0, j=5 ; i<10 ; i++, j=j+5 )
            {
            printf( "Hola " );
            printf( "Esta es la línea %i", i );
            printf( "j vale = %i\n", j );
            }
    }
```

Como vemos en el ejemplo tenemos más de una variable en la sección de inicialización y en la de incrementos. También podíamos haber puesto más de una condición. Los elementos de cada sección se separan por comas. Cada sección se separa por punto y coma.

Más tarde veremos cómo usar el for con cadenas.

[Arriba]

While

El formato del bucle while es es siguiente:

```
    while ( condición )
            {
            bloque de instrucciones a ejecutar
            }
```

While quiere decir *mientras*. Aquí se ejecuta el bloque de intrucciones mientras se cumpla la condición impuesta en while.

Vamos a ver un ejemplo:

```
    #include <stdio.h>

    int main()
            {
            int contador = 0;

            while ( contador<100 )
                    {
                    contador++;
                    printf( "Ya voy por el %i,
pararé enseguida.\n", contador );
                    }
            }
```

Este programa imprime en pantalla los valores del 1 al 100. Cuando i=100 ya no se cumple la condición. Una cosa importante, si hubíesemos cambiado el orden de las instrucciones a ejecutar:

```
    ...
    printf( "Ya voy por el %i, pararé
enseguida.\n", contador );
    contador++;
    ...
```

En esta ocasión se imprimen los valores del 0 al 99. Cuidado con esto, que a veces produce errores difíciles de encontrar.

[Arriba]

Do While

El formato del bucle do-while es:

```
    do
            {
            instrucciones a ejecutar
            } while ( condición );
```

La diferencia entre *while* y *do-while* es que en este último, la condición va despues del conjunto de instrucciones a ejecutar. De esta forma, esas instrucciones se ejecutan al menos una vez.

Su uso es similar al de while.

[Arriba]

Sentencias de condición

Hasta aquí hemos visto cómo podemos repetir un conjunto de instrucciones las veces que deseemos. Pero ahora vamos a ver cómo podemos controlar totalmente el flujo de un programa. Dependiendo de los valores de alguna variable se tomarán unas acciones u otras. Empecemos con la sentencia *if*.

[Arriba]

If

La palabra *if* significa *si* (condicional), pero supongo que esto ya lo sabías. Su formato es el siguiente:

```
    if ( condición )
            {
            instrucciones a ejecutar
            }
```

Cuando se cumple la condición entre paréntesis se ejecuta el bloque inmediatamente siguiente al if (bloque *instrucciones a ejecutar*).

En el siguiente ejemplo tenemos un programa que nos pide un número, si ese número es 10 se muestra un mensaje. Si no es 10 no se muestra ningún mensaje:

```
    #include <stdio.h>

    int main()
            {
            int num;

            printf( "Introduce un número " );
            scanf( "%i", &num );
            if (num==10)
                    {
                    printf( "El número es
correcto\n" );
                    }
            }
```

Como siempre, la condición es falsa si es igual a cero. Si es distinta de cero será verdadera.

[Arriba]

If - Else

El formato es el siguiente:

```
    if ( condición )
            {
            bloque que se ejecuta si se cumple la
condición
            }
    else
            {
            bloque que se ejecuta si no se cumple
la condición
            }
```

En el if si no se cumplía la condición no se ejecutaba el bloque siguiente y el programa seguía su curso normal. Con el if else tenemos un bloque adicional que sólo se ejecuta si no se cumple la condición. Veamos un ejemplo:

```
    #include <stdio.h>

    int main()
            {
            int a;
            printf( "Introduce un número " );
            scanf( "%i", &a );
            if ( a==8 )
                    {
                    printf ( "El número introducido
era un ocho.\n" );
                    }
            else
                    {
                    printf ( "Pero si no has
escrito un ocho!!!"\n" );
                    }
            }
```

Al ejecutar el programa si introducimos un 8 se ejecuta el bloque siguiente al if y se muestra el mensaje:

```
El número introducido era un ocho.
```

Si escribimos cualquier otro número se ejecuta el bloque siguiente al else mostrándose el mensaje:

```
Pero si no has escrito un ocho!!!
```

[Arriba]

If else if

Se pueden poner if else anidados si se desea:

```
    #include <stdio.h>

    int main()
            {
            int a;

            printf( "Introduce un número " );
            scanf( "%i", &a );
            if ( a<10 )
```

```
                    {
                    printf ( "El número introducido
era menor de 10.\n" );
                    }
            else if ( a>10 && a<100 )
                    {
                    printf ( "El número está entre
10 y 100"\n" );
                    }
            else if ( a>100 )
                    {
                    printf( "El número es mayor que
100\n" );
                    }
            printf( "Fin del programa\n" );
            }
```

El símbolo **&&** de la condición del segundo if es un AND (Y). De esta forma la condición queda: *Si a es mayor que 10 Y a es menor que 100*. Cosulta la sección **Notas sobre condiciones** para saber más.

Y así todos los if else que queramos. Si la condición del primer if es verdadera se muestra el mensaje *El número introducido era menor de 10* y se saltan todos los if-else siguientes (se muestra *Fin del programa*). Si la condición es falsa se ejecuta el siguiente else-if y se comprueba si a está entre 10 y 100. Si es cierto se muestra *El número está entre 10 y 100*. Si no es cierto se evalúa el último else-if.

[Arriba]

? (el otro if-else)

El uso de la interrogación es una forma de condensar un if-else. Su formato es el siguiente:

```
    ( condicion ) ? ( instrucción 1 ) : (
instrucción 2 )
```

Si se cumple la condición se ejecuta la *instrucción 1* y si no se ejecuta la *instrucción 2*. Veamos un ejemplo con el if-else y luego lo reescribimos con "?":

```
    #include <stdio.h>
```

```
    int main()
            {
            int a;
            int b;

            printf( "Introduce un número " );
            scanf( "%i", &a );
            if ( a<10 )
                    {
                    b = 1;
                    }
            else
                    {
                    b = 4;
                    }
            printf ( "La variable 'b' toma el
valor: %i\n", b );
            }
```

Si el valor que tecleamos al ejecutar es menor que 10 entonces b=1, en cambio si tecleamos un número mayor que 10 'b' será igual a 2. Ahora vamos a reescribir el programa usando '?':

```
    #include <stdio.h>

    int main()
            {
            int a;
            int b;

            printf( "Introduce un número " );
            scanf( "%i", &a );
            b = ( a<10 ) ? 1 : 4 ;
            printf ( "La variable 'b' toma el
valor: %i\n", b );
            }
```

¿Qué es lo que sucede ahora? Se evalúa la condición a<10. Si es verdadera (a menor que 10) se ejecuta la instrucción 1, es decir, que b toma el valor '1'. Si es falsa se ejecuta la instrucción 2, es decir, b toma el valor '4'.

Esta es una sentencia muy curiosa pero sinceramente creo que no la he usado casi nunca en mis programas y tampoco la he visto mucho en programas ajenos.

[Arriba]

Switch

El formato de la sentencia switch es:

```
    switch ( variable )
            {
            case opción 1:
                    código a ejecutar si la
variable tiene el
                    valor de la opción 1
                    break;
            case opción 2:
                    código a ejecutar si la
variable tiene el
                    valor de la opción 2
                    break;
            default:
                    código que se ejecuta si la
variable tiene
                    un valor distinto a los
anteriores
                    break;
            }
```

Vamos a ver cómo funciona. La sentencia switch sirve par elegir una opción entre varias disponibles. Aquí no tenemos una condición que se debe cumplir sino el valor de una variable. Dependiendo del valor se cumplirá un caso u otro.

Vamos a ver un ejemplo de múltiples casos con if-else y luego con switch:

```
    #include <stdio.h>

    int main()
            {
            int num;

            printf( "Introduce un número " );
            scanf( "%i", &num );
            if ( num==1 )
                    printf ( "Es un 1\n" );
            else if ( num==2 )
                    printf ( "Es un 2\n" );
            else if ( num==3 )
                    printf ( "Es un 3\n" );
            else
                    printf ( "No era ni 1, ni 2, ni
3\n" );
            }
```

Ahora con switch:

```
    #include <stdio.h>

    int main()
            {
            int num;

            printf( "Introduce un número " );
            scanf( "%i", &num );
            switch( num )
                    {
                    case 1:
                            printf( "Es un 1\n" );
                            break;
                    case 2:
                            printf( "Es un 2\n" );
                            break;
                    case 3:
                            printf( "Es un 3\n" );
                            break;
                    default:
                            printf( "No es ni 1, ni
2, ni 3\n" );
                    }
            }
```

Como vemos el código con switch es más cómodo de leer.

Vamos a ver qué pasa si nos olvidamos algún break:

```
    #include <stdio.h>

    int main()
            {
            int num;

            printf( "Introduce un número " );
            scanf( "%i", &num );
            switch( num )
                    {
                    case 1:
                            printf( "Es un 1\n" );
                            /* Nos olvidamos el
break que debería haber aquí */
                    case 2:
                            printf( "Es un 2\n" );
                            break;
                    default:
                            printf( "No es ni 1, ni
2, ni 3\n" );
                    }
            }
```

Si al ejecutar el programa escribimos un dos tenemos el mensaje *Es un dos*. Todo correcto. Pero si escribimos un uno lo que nos sale en pantalla es:

```
    Es un 1
    Es un 2
```

¿Por qué? Pues porque cada caso empieza con un case y acaba donde hay un break. Si no ponemos break aunque haya un case el programa sigue hacia adelante. Por eso se ejecuta el código del case 1 y del case 2.

Puede parecer una desventaja pero a veces es conveniente. Por ejemplo cuando dos case deben tener el mismo código. Si no tuviéramos esta posibilidad tendríamos que escribir dos veces el mismo código. (Vale, vale, también podríamos usar funciones, pero si el código es corto puede ser más conveniente no usar funciones. Ya hablaremos de eso más tarde.).

Sin embargo switch tiene algunas limitaciones, por ejemplo no podemos usar condiciones en los case. El ejemplo que hemos visto en el apartado if-else-if no podríamos hacerlo con switch.

[Arriba]

Sentecias de salto

Goto

La sentencia goto (*ir a*) nos permite hacer un salto a la parte del programa que deseemos. En el programa podemos poner etiquetas, estas etiquetas no se ejecutan. Es como poner un nombre a una parte del programa. Estas etiquetas son las que nos sirven para indicar a la sentencia goto dónde tiene que saltar.

```
    #include <stdio.h>

    int main()
            {
            printf( "Línea 1\n" );
            goto linea3;   /* Le decimos al goto
que busque la etiqueta linea3 */
            printf( "Línea 2\n" );
            linea3:                /* Esta es la
etiqueta */
```

```
            printf( "Línea 3\n" );
            }
```

Resultado:

```
Línea 1
Línea 3
```

Como vemos no se ejecuta el printf de Línea 2 porque nos lo hemos saltado con el goto.

El goto sólo se puede usar dentro de funciones, y no se puede saltar desde una función a otra. (Las funciones las estudiamos en el siguiente capítulo).

Un apunte adicional del goto: Cuando yo comencé a programar siempre oía que no era correcto usar el goto, que era una mala costumbre de programación. Decían que hacía los programas ilegibles, difíciles de entender. Ahora en cambio se dice que no está tan mal. Yo personalmente me he encontrado alguna ocasión en la que usar el goto no sólo no lo hacía ilegible sino que lo hacía más claro. En Internet se pueden encontrar páginas que discuten sobre el tema. Pero como conclusión yo diría que cada uno la use si quiere, el caso es no abusar de ella y tener cuidado.

[Arriba]

Notas sobre las condiciones

Las condiciones de las sentencias se evalúan al ejecutarse. De esta evaluación obtenemos un número. Las condiciones son falsas si este número es igual a cero. Son verdaderas si es distinto de cero (los números negativos son verdaderos).

Ahí van unos ejemplos:

```
    a = 2;
    b = 3;
    if ( a == b ) ...
```

Aquí **a==b** sería igual a 0, luego falso.

```
    if ( 0 ) ...
```

Como la condición es igual a cero, es falsa.

```
    if ( 1 ) ...
```

Como la condición es distinta de cero, es verdadera.

```
    if ( -100 ) ...
```

Como la condición es distinta de cero, es verdadera.

Supongamos que queremos mostrar un mensaje si una variable es distinta de cero:

```
    if ( a!=0 ) printf( "Hola\n" );
```

Esto sería redundante, bastaría con poner:

```
    if ( a ) printf( "Hola\n" );
```

Esto sólo vale si queremos comprobar que es distinto de cero. Si queremos comprobar que es igual a 3:

```
    if ( a == 3 ) printf( "Es tres\n" );
```

Como vemos las condiciones no sólo están limitadas a comparaciones, se puede poner cualquier función que devuelva un valor. Dependiendo de si este valor es cero o no, la condición será falsa o verdadera.

También podemos probar varias condiciónes en una sola usando && (AND), || (OR).

Ejemplos de && (AND):

```
if ( a==3 && b==2 ) printf( "Hola\n" );   /* Se
cumple si a es 3 Y b es dos */

if ( a>10 && a<100 ) printf( "Hola\n" );  /* Se
cumple si a es mayor

que 10 Y menor que 100 */

if ( a==10 && b<300 ) printf( "Hola\n" ); /* Se
cumple si a es igual a 10

Y b es menor que 300 */
```

Ejemplos de || (OR):

```
    if ( a<100 || b>200 ) printf( "Hola\n" ); /*
Se cumple si a menor que 100

O b mayor que 200 */
    if ( a<10 || a>100 ) printf( "Hola\n" ); /* Se
cumple si a menor que

10 O a mayor que 100 */
```

Se pueden poner más de dos condiciones:

```
    if ( a>10 && a<100 && b>200 && b<500 )/* Se
deben cumplir las cuatro condiciones */
```

Esto se cumple si a está entre 10 y 100 y b está entre 200 y 500.

También se pueden agrupar mediante paréntesis varias condiciones:

```
    if (  ( a>10 && a<100 ) || ( b>200 && b<500 )
)
```

Esta condición se leería como sigue:

```
    si a es mayor que 10 y menor que 100
                  o
    si b es mayor que 200 y menor que 500
```

Es decir que si se cumple el primer paréntesis o si se cumple el segundo la condición es cierta.

[Arriba]

Ejercicios

Ejercicio 1: ¿Cuántas veces nos pide el siguiente programa un número y por qué?

```
#include <stdio.h>



int main() {

    int i;

    int numero, suma = 0;



    for ( i=0; i<4; i++ );

        {

        printf( "\nIntroduce un número: " );

        scanf( "%d", &numero );

        suma += numero;

        }

    printf ( "\nTotal: %d\n", suma );

    system( "PAUSE" );

}
```

Solución: El programa se ejecutará una única vez puesto que al final de la sentencia for hay un punto y coma. Como sabemos, el bucle for hace que se ejecuten las veces necesarias la sentencia siguiente (o el siguiente bloque entre {}). Para que el programa funcione correctamente habría que eliminar el punto y coma.

Ejercicio 2: Una vez eliminado el punto y coma ¿cuántas veces nos pide el programa anterior un número?

Solución: Se ejecuta cuatro veces. Desde i=0 mientras la segunda condición sea verdadera, es decir, desede i=0 hasta i=3.

Ejercicio 3: Escribe un programa que muestre en pantalla lo siguiente:

```
*

**

***

****

*****
```

Solución:

```
#include <stdio.h>



int main() {

    int i, j;



    for( i=0; i<6; i++ ) {

        for( j=0; j<i; j++ )

                printf( "*" );
```

```
        printf( "\n" );

    }

    return 0;

}
```

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Introducción a las Funciones

Contenido del Capítulo:

Introducción

Vamos a dar un paso más en la complejidad de nuestros programas. Vamos a empezar a usar funciones creadas por nosotros. Las funciones son de una gran utilidad en los programas. Nos ayudan a que sean más legibles y más cortos. Con ellos estructuramos mejor los programas.

Una función sirve para realizar tareas concretas y simplificar el programa. Nos sirve para evitar tener que escribir el mismo código varias veces. Ya hemos visto en el curso algunas funciones como printf, gotoxy, scanf y clrscr. Algunas de éstas están definidas en una biblioteca (la biblioteca estándar de C). No tenemos que preocuparnos de ella porque el compilador se encarga de ella automáticamente.

Sin embargo nosotros también podemos definir nuestras propias funciones. Pocas veces se ve un programa un poco complejo que no use funciones. Una de ellas, que usamos siempre, es la función main.

[Arriba]

Definición de una función

Una función tiene el siguiente formato:

```
tipo_de_variable nombre_de_la_función( argumentos
)
     {
     definición de variables;
```

```
    cuerpo de la función;

    return 0;
    }
```

Poco a poco, empecemos por el nombre_de_la_función. Para el nombre no se pueden usar mas que letras, números y el símbolo '_'. No se pueden usar ni acentos ni espacios. Además el nombre de la función debe empezar por una letra, no puede empezar con un número. El nombre de la función se usa para llamarla dentro del programa.

El tipo_de_variable: Cuando una función se ejecuta y termina debe devolver un valor. Este valor puede ser cualquiera de los tipos de variables que hemos visto en el capítulo de Tipos de datos (int, char, float, double) o un tipo de dato definido por nosotros (esto lo veremos más tarde). El valor que devuelve la función suele ser el resultado de las operaciones que se realizan en la función, o si han tenido éxito o no.

También podemos usar el tipo void. Este nos permite que podamos devolver cualquier tipo de variable o ninguna.

Definición de variables: Dentro de la función podemos definir variables que sólo tendrán validez dentro de la propia función. Si declaramos una variable en una función no podemos usarla en otra.

Cuerpo de la función: Aquí es donde va el código de la función.

Return: Antes hemos indicado que la función devuelve un valor. La sentencia return se usa para esto. El dato que se pone despues de return es el dato que se devuelve. Puede ser una constante o una variable. Debe ser del mismo tipo que *tipo_de_variable*.

Argumentos: Estos son variables que se pasan como datos a una función. Deben ir separados por una coma. Cada variable debe ir con su tipo de variable.

Las funciones deben definirse antes de ser llamadas. En los ejemplos a continuación se llama a la función desde main, así que tenemos que definirlas antes que main. Esto lo veremos en el apartado siguiente.

Ejemplo 1. Función sin argumentos que no devuelve nada:

Este programa llama a la función prepara pantalla que borra la pantalla y muestra el mensaje "la pantalla está limpia". Por supuesto es de nula utilidad pero nos sirve para empezar.

```
#include <stdio.h>
#include <conio.h>

void prepara_pantalla()     /* No se debe poner
punto y coma aquí */
     {
     clrscr();
     printf( "La pantalla está limpia\n" );
     return;     /* No hace falta devolver ningún
valor, mucha gente ni siquiera pone este return */
     }

int main()
     {
     prepara_pantalla();/* Llamamos a la función
*/
     }
```

Ejemplo 2. Función con argumentos, no devuelve ningún valor:

En este ejemplo la función **compara** toma dos números, los compara y nos dice cual es mayor.

```
#include <stdio.h>
#include <conio.h>

void compara( int a, int b )     /* Metemos los
parámetros a y b a la función */
     {
     if ( a>b ) printf( "%i es mayor que %i\n" ,
a, b );
     else printf( "%i es mayor que %i\n", b, a );
     }

int main()
     {
     int num1, num2;

     printf( "Introduzca dos números: " );
     scanf( "%i %i", &num1, &num2 );

     compara( num1, num2 );/* Llamamos a la
función con sus dos argumentos */

     return 0;
     }
```

Ejemplo 3. Función con argumentos que devuelve un valor.

Este ejemplo es como el anterior pero devuelve como resultado el mayor de los dos números.

```
#include <stdio.h>
#include <conio.h>

int compara( int a, int b )     /* Metemos los
parámetros a y b a la función */
     {
     int mayor;     /* Esta función define su
propia variable, esta variable sólo se puede usar
aquí */

     if ( a>b )
     mayor = a;
     else mayor = b;

     return mayor;
     }

int main()
     {
     int num1, num2;
     int resultado;

     printf( "Introduzca dos números: " );
     scanf( "%i %i", num1, num2 );

     resultado = compara( num1, num2 );/*
Recogemos el valor que devuelve la función en
resultado */
     printf( "El mayor de los dos es %i\n",
resultado );

     return 0;
     }
```

En este ejemplo podíamos haber hecho también:

```
     printf( "El mayor de los dos es %i\n",
compara( num1, num2 ) );
```

De esta forma nos ahorramos tener que definir la variable 'resultado'.

[Arriba]

Dónde se definen las funciones

Las funciones deben definirse siempre antes de donde se usan. Lo habitual en un programa es:

Sección	Descripción
Includes	Aquí se indican qué ficheros externos se usan
Definiciones	Aquí se definen las constantes que se usan en el programa
Definición de variables	Aquí se definen las variables globales (las que se pueden usar en TODAS las funciones)
Definición de funciones	Aquí es donde se definen las funciones
Función main	Aquí se define la función main.

Esta es una forma muy habitual de estructurar un programa. Sin embargo esto no es algo rígido, no tiene por qué hacerse así, pero es recomendable.

Se puede hacer de otra forma, también aconsejable. Consiste en definir después de las variables las cabeceras de las funciones, sin escribir su código. Esto nos permite luego poner las fucniones en cualquier orden. Ejemplos:

```
#include <stdio.h>
#include <conio.h>

void compara( int a, int b );     /* Definimos la
cabecera de la función */

int main()
     {
     int num1, num2;
     int resultado;

     printf( "Introduzca dos números: " );
     scanf( "%i %i", num1, num2 );

     resultado = compara( num1, num2 );
     printf( "El mayor de los dos es %i\n",
resultado );

     return 0;
     }

int compara( int a, int b )     /* Ahora podemos
poner el cuerpo de la función donde queramos. */
               /* Incluso después de donde la
llamamos (main) */
     {
     int mayor;
     if ( a>b )
     mayor = a;
     else mayor = b;

     return mayor;
     }
```

Cuando se define la cabecera de la función sin su cuerpo (o código) debemos poner un ';' al final. Cuando definamos el cuerpo más tarde no debemos poner el ';', se hace como una función normal.

La definición debe ser igual cuando definimos sólo la cabecera y cuando definimos el cuerpo. Mismo nombre, mismo número y tipo de parámetros y mismo tipo de valor devuelto.

[Arriba]

Vida de una variable

Cuando definimos una variable dentro de una función, esa variable sólo es válida dentro de la función. Si definimos una variable dentro de main sólo podremos usarla dentro de main. Si por el contrario la definimos como variable global, antes de las funciones, se puede usar en todas las funciones.

Podemos crear una variable global y en una función una variable local con el mismo nombre. Dentro de la función cuando llamamos a esa variable llamamos a la local, no a la global. Esto no da errores pero puede crear confusión al programar y al analizar el código.

[Arriba]

Ejercicios

Ejercicio 1: Descubre los errores:

```
#include <stdio.h>
#include <conio.h>

int main()
     {
     int num1, num2;
     int resultado,

     printf( "Introduzca dos números: " );
     scanf( "%i %i", &num1, &num2 );
     resultado = compara( num1, num2 );
     printf( "El mayor de los dos es %i\n",
resultado );

     return 0;
     }
```

```
int compara( int a, int b );
     {
     int mayor;
     if ( a>b )mayor = a;
     else mayor = b;
     return mayor;
     }
```

Solución:

- Hay una coma después de int resultado en vez de un punto y coma.
- Llamamos a la función compara dentro de main antes de definirla.
- Cuando definimos la función compara con su cuerpo hemos puesto un punto y coma al final, eso es un error.

Ejercicio 2: Busca los errores.

```
#include <stdio.h>

int resultado( int parametro )

int main()
     {
     int a, b;

     a = 2; b = 3;
     printf( "%i", resultado( a );

     return 0;
     }

char resultado( int parametro )
     {
     return parámetro+b;
     }
```

Solución:

- Hemos definido la cabecera de resultado sin punto y coma.
- Cuando definimos el cuerpo de resultado en su cabecera hemos puesto char, que no coincide con la definición que hemos hecho al principio.
- En la función resultado estamos usando la variable 'b' que está definida sólo en main. Por lo tanto es como si no existiera para resultado.
- En printf nos hemos dejado un paréntesis al final.

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Punteros

Contenido del Capítulo:

Introducción

Este capítulo puede resultar problemático a aquellos que no han visto nunca lo que es un puntero. Por lo tanto si tienes alguna duda o te parece que alguna parte está poco clara ponte en contacto conmigo.

Punteros!!! uff. Si hasta ahora te había parecido complicado prepárate. Este es uno de los temas que más suele costar a la gente al aprender C. Los punteros son una de las más potentes características de C, pero a la vez uno de sus mayores peligros. Los punteros nos permiten acceder directamente a cualquier parte de la memoria. Esto da a los programas C una gran potencia. Sin embargo son una fuente ilimitada de errores. Un error usando un puntero puede bloquear el sistema (si usamos ms-dos o win95, no en Linux) y a veces puede ser difícil detectarlo.

Otros lenguajes no nos dejan usar punteros para evitar estos problemas, pero a la vez nos quitan parte del control que tenemos en C. No voy a entrar a discutir si es mejor o no poder usar punteros, aunque pienso que es mejor. Yo me voy a limitar a explicar cómo funcionan.

A pesar de todo esto no hay que tenerles miedo. Casi todos los programas C usan punteros. Si aprendemos a usarlos bien no tendremos mas que algún problema esporádico. Así que atención, valor y al toro.

La memoria del ordenador

Si tienes bien claro lo que es la memoria del ordenador puedes saltarte esta sección. Pero si confundes la memoria con el disco duro o no tienes claro lo que es no te la pierdas.

A lo largo de mi experiencia con ordenadores me he encontrado con mucha gente que no tiene claro cómo funciona un ordenador. Cuando hablamos de memoria nos estamos refiriendo a la memoria RAM del ordenador. Son unas *pastillas* que se conectan a la placa base y nada tienen que ver con el disco duro. El disco duro guarda los datos permanentemente (hasta que se rompe) y la información se almacena como ficheros. Nosotros podemos decirle al ordenador cuándo grabar, borrar, abrir un documento, etc. La memoria Ram en cambio, se borra al apagar el ordenador. La memoria Ram la usan los programas sin que el usuario de éstos se de cuenta.

Para hacernos una idea, hoy en dia la memoria se mide en MegaBytes (suelen ser 16, 32, 64, 128Mb) y los discos duros en GigaBytes (entre 3,4 y 70Gb, o mucho más).

Hay otras memorias en el ordenador aparte de la mencionada. La memoria de video (que está en la tarjeta gráfica), las memorias caché (del procesador, de la placa...) y quizás alguna más que ahora se me olvida.

[Arriba]

Direcciones de variables

Vamos a ir como siempre por partes. Primero vamos a ver qué pasa cuando declaramos una variable.

Al declarar una variable estamos diciendo al ordenador que nos reserve una parte de la memoria para almacenarla. Cada vez que ejecutemos el programa la variable se almacenará en un sitio diferente, eso no lo podemos controlar, depende de la memoria disponible y otros factores misteriosos. Puede que se almacene en el mismo sitio, pero es mejor no fiarse. Dependiendo del tipo de variable que declaremos el ordenador nos reservará más o menos memoria. Como vimos en el capítulo de tipos de datos cada tipo de variable ocupa más o menos bytes. Por ejemplo si declaramos un char, el ordenador nos reserva 1 byte (8 bits). Cuando finaliza el programa todo el espacio reservado queda libre.

Existe una forma de saber qué direcciones nos ha reservado el ordenador. Se trata de usar el operador **&** (operador de dirección). Vamos a ver un ejemplo: Declaramos la variable 'a' y obtenemos su valor y dirección.

```
#include <stdio.h>

int main()
     {
     int a;

     a = 10;
     printf( "Dirección de a = %p, valor de a =
%i\n", &a, a );
     }
```

Para mostrar la dirección de la variable usamos %p en lugar de %i, sirve para escribir direcciones de punteros y variables. El valor se muestra en hexadecimal.

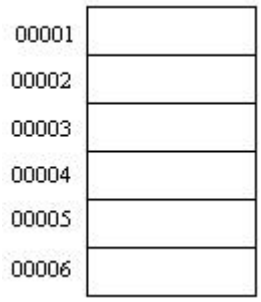
No hay que confundir el valor de la variable con la dirección donde está almacenada la variable. La variable 'a' está almacenada en un lugar determinado de la memoria, ese lugar no cambia mientras se ejecuta el programa. El valor de la variable puede cambiar a lo largo del programa, lo cambiamos nosotros. Ese valor está almacenado en la dirección de la variable. El nombre de la variable es equivalente a poner un nombre a una zona de la memoria. Cuando en el programa escribimos 'a', en realidad estamos diciendo, "el valor que está almacenado en la dirección de memoria a la que llamamos 'a'".

[Arriba]

Que son los punteros

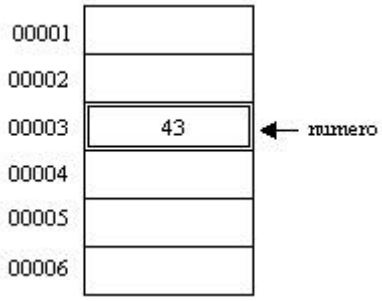
Ahora ya estamos en condiciones de ver lo que es un puntero. Un puntero es una variable un tanto especial. Con un puntero podemos almacenar direcciones de memoria. En un puntero podemos tener guardada la dirección de una variable.

Vamos a ver si cogemos bien el concepto de puntero y la diferencia entre éstos y las variables *normales*.



En el dibujo anterior tenemos una representación de lo que sería la memoria del ordenador. Cada casilla representa un byte de la memoria. Y cada número es su dirección de memoria. La primera casilla es la posición 00001 de la memoria. La segunda casilla la posición 00002 y así sucesivamente.

Supongamos que ahora declaramos una variable char: ***char numero = 43***. El ordenador nos guardaría por ejemplo la posición 00003 para esta variable. Esta posición de la memoria queda reservada y ya no la puede usar nadie más. Además esta posición a partir de ahora se le llama *numero*. Como le hemos dado el valor 43 a numero, el valor 43 se almacena en numero, es decir, en la posición 00003.



Si ahora usáramos el programa anterior:

```
#include <stdio.h>

int main()
     {
     int numero;

     numero = 43;
     printf( "Dirección de numero = %p, valor de
numero = %i\n", &numero, numero );
     }
```

El resultado sería:

```
Dirección de numero = 00003, valor de numero = 43
```

Creo que así ya está clara la diferencia entre el valor de una variable (43) y su dirección (00003). Ahora vamos un poco más allá, vamos a declarar un puntero. Hemos dicho que un puntero sirve para almacenar la direcciones de memoria. Muchas veces los punteros se usan para guardar las direcciones de variables. Vimos en el capítulo Tipos de Datos que cada tipo de variable ocupaba un espacio distinto. Por eso cuando declaramos un puntero debemos especificar el tipo de datos cuya dirección almacenará. En nuestro ejemplo queremos que almacene la dirección de una variable char. Así que para declarar el puntero **punt** debemos hacer:

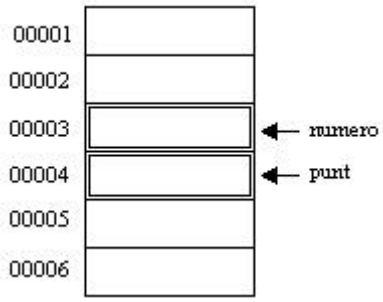
```
char *punt;
```

El * (asterisco) sirve para indicar que se trata de un puntero, debe ir justo antes del nombre de la variable, sin espacios. En la variable punt sólo se pueden guardar direcciones de memoria, no se pueden guardar datos. Vamos a volver sobre el ejemplo anterior un poco ampliado para ver cómo funciona un puntero:

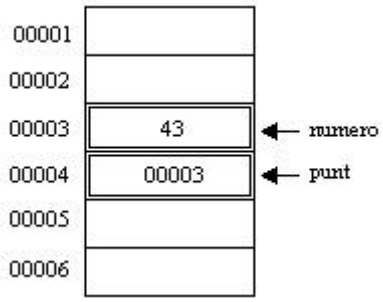
```
#include <stdio.h>

int main()
     {
     int numero;
     int *punt;

     numero = 43;
     punt = &numero;
     printf( "Dirección de numero = %p, valor de
numero = %i\n", &numero, numero );
     }
```

Vamos a ir línea a línea:

- En la primera *int numero* reservamos memoria para numero (supongamos que queda como antes, posición 00003). Por ahora numero no tiene ningún valor.
- Siguiente línea: *int *punt;*. Reservamos una posición de memoria para almacenar el puntero. Lo normal es que según se declaran variables se guarden en posiciones contiguas. De modo que quedaría en la posición 00004. Por ahora punt no tiene ningún valor, es decir, no apunta a ninguna variable. Esto es lo que tenemos por ahora:



- Tercera línea: *numero = 43;*. Aquí ya estamos dando el valor 43 a numero. Se almacena 43 en la dirección 00003, que es la de numero.
- Cuarta línea: *punt = №*. Por fin damos un valor a punt. El valor que le damos es la dirección de numero (ya hemos visto que & devuelve la dirección de una variable). Así que punt tendrá como valor la dirección de numero, 00003. Por lo tanto ya tenemos:



Cuando un puntero tiene la dirección de una variable se dice que ese puntero **apunta** a esa variable.

NOTA: La declaración de un puntero depende del tipo de dato al que queramos apuntar. En general la declaración es:

```
tipo_de_dato *nombre_del_puntero;
```

Si en vez de querer apuntar a una variable tipo char como en el ejemplo hubiese sido de tipo int:

```
int *punt;
```

[Arriba]

Para qué sirve un puntero y cómo se usa

Los punteros tienen muchas utilidades, por ejemplo nos permiten pasar argumentos (o parámetros) a una función y modificarlos. También permiten el manejo de cadenas y de arrays. Otro uso importante es que nos permiten acceder directamente a la pantalla, al teclado y a todos los componenetes del ordenador. Pero esto ya lo veremos más adelante.

Pero si sólo sirvieran para almacenar direcciones de memoria no servirían para mucho. Nos deben dejar también la posibilidad de acceder a esas posiciones de memoria. Para acceder a ellas se usa el operador *, que no hay que confundir con el de la multiplicación.

```
#include <stdio.h>

int main()
     {
     int numero;
     int *punt;

     numero = 43;
     punt = &numero;
     printf( "Dirección de numero = %p, valor de
numero = %i\n", &numero, *punt );
     }
```

Si nos fijamos en lo que ha cambiado con respecto al ejemplo anterior, vemos que para acceder al valor de número usamos *punt en vez de numero. Esto es así porque punt apunta a numero y *punt nos permite acceder al valor al que apunta punt.

```
#include <stdio.h>

int main()
     {
     int numero;
     int *punt;

     numero = 43;
     punt = &numero;
     *punt = 30;
     printf( "Dirección de numero = %p, valor de
numero = %i\n", &numero, numero );
     }
```

Ahora hemos cambiado el valor de numero a través de *punt.

En resumen, usando punt podemos apuntar a una variable y con *punt vemos o cambiamos el contenido de esa variable.

Un puntero no sólo sirve para apuntar a una variable, también sirve para apuntar una dirección de memoria determinada. Esto tiene muchas

aplicaciones, por ejemplo nos permite controlar el hardware directamente (en MS-Dos y Windows, no en Linux). Podemos escribir directamente sobre la memoria de video y así escribir directamente en la pantalla sin usar printf.

[Arriba]

Usando punteros en una comparación

Veamos el siguiente ejemplo. Queremos comprobar si dos variables son iguales usando punteros:

```
#include <stdio.h>

int main()
     {
     int a, b;
     int *punt1, *punt2;

     a = 5; b = 5;
     punt1 = &a; punt2 = &b;

     if ( punt1 == punt2 )
     printf( "Son iguales\n" );
     }
```

Alguien podría pensar que el *if* se cumple y se mostraría el mensaje *Son iguales* en pantalla. Pues no es así, el programa es erróneo. Es cierto que a y b son iguales. También es cierto que punt1 apunta a 'a' y punt2 a 'b'. Lo que queríamos comprobar era si a y b son iguales. Sin embargo con la condición estamos comprobando si punt1 apunta al mismo sitio que punt2, estamos comparando las direcciones donde apuntan. Por supuesto a y b están en distinto sitio en la memoria así que la condición es falsa. Para que el programa funcionara deberíamos usar los asteriscos:

```
#include <stdio.h>

int main()
     {
     int a, b;
     int *punt1, *punt2;

     a = 5; b = 5;
     punt1 = &a; punt2 = &b;

     if ( *punt1 == *punt2 )
     printf( "Son iguales\n" );
     }
```

Ahora sí. Estamos comparando el contenido de las variables a las que apuntan punt1 y punt2. Debemos tener mucho cuidado con esto porque es un error que se cuela con mucha facilidad.

Vamos a cambiar un poco el ejemplo. Ahora 'b' no existe y punt1 y punt2 apuntan a 'a'. La condición se cumplirá porque apuntan al mismo sitio.

```
#include <stdio.h>

int main()
     {
     int a;
     int *punt1, *punt2;

     a = 5;
     punt1 = &a; punt2 = &a;

     if ( punt1 == punt2 )
     printf( "punt1 y punt2 apuntan al mismo
sitio\n" );
     }
```

[Arriba]

Punteros como argumentos de funciones

Hemos visto en el capítulo de funciones cómo pasar parámetros y cómo obtener resultados de las funciones (con los valores devueltos con return). Pero tiene un inconveniente, sólo podemos tener un valor devuelto. Ahora vamos a ver cómo los punteros nos permites modificar varias variables en una función.

Hasta ahora para pasar una variable a una función hacíamos lo siguiente:

```
#include <stdio.h>

int suma( int a, int b )
     {
     return a+b;
     }

int main()
     {
     int var1, var2, resultado;

     var1 = 5; var2 = 8;
     resultado = suma(var1, var2);
```

```
     printf( "La suma es : %i\n", resultado );
     }
```

Bien aquí hemos pasado a la función los parámetros 'a' y 'b' (que no podemos modificar) y nos devuelve la suma de ambos. Supongamos ahora que queremos tener la suma pero además queremos que var1 se haga cero dentro de la función. Para eso haríamos lo siguiente:

```
#include <stdio.h>

int suma( int *a, int b )
     {
     int c;

     c = *a + b;
     *a = 0;
     return c;
     }

int main()
     {
     int var1, var2, resultado;

     var1 = 5; var2 = 8;
     resultado = suma(&var1, var2);
     printf( "La suma es: %i y a vale: %i\n",
resultado , var1 );
     }
```

Fijémonos en lo que ha cambiado (con letra en negrita): En la función suma hemos declarado 'a' como puntero. En la llamada a la función (dentro de main) hemos puesto & para pasar la dirección de la variable var1. Ya sólo queda hacer cero a var1 a través de *a=0.

También usamos una variable 'c' que nos servirá para almacenar la suma de 'a' y 'b'.

Es importante no olvidar el operador & en la llamada a la función ya que sin el no estaríamos pasando la dirección de la variable sino cualquier otra cosa.

Podemos usar tantos punteros como queramos en la definición de la función.

NOTA IMPORTANTE: Existe la posibilidad de hacer el ejercicio de esta otra manera, sin usar la variable *resultado* (de hecho, en una versión anterior del curso estaba así):

```
#include <stdio.h>

int suma( int *a, int b )
     {
     int c;

     c = *a + b;
     *a = 0;
     return c;
     }

int main()
     {
     int var1, var2;

     var1 = 5; var2 = 8;
     printf( "La suma es: %i y a vale: %i\n",
suma(&var1, var2) , var1 );
     }
```

Sin embargo, esto puede dar problemas, ya que no podemos asegurar de cómo va a evaluar el compilador los argumentos de printf. Es posible que primero almacene el varlor de var1 antes de evaluar *suma*. Si ocurriese así el resultado del programa sería: *La suma es 13 y a vale 5*, en lugar de La suma es 13 y a vale 0.

[Arriba]

Ejercicios

Ejercicio 1: Encuentra un fallo muy grave:

```
#include <stdio.h>

int main()
     {
     int *a;

     *a = 5;
     }
```

Solución: No hemos dado ninguna dirección al puntero. No sabemos a dónde apunta. Puede apuntar a cualquier sitio, al darle un valor estamos encribiendo en un lugar desconocido de la memoria. Esto puede dar problemas e incluso bloquear el ordenador. Recordemos que al ejecutar un programa éste se copia en la memoria, al escribir en cualquier parte puede que estemos cambiando el programa (en la memoria, no en el disco duro).

Ejercicio 2: Escribe un programa que de un valor a una variable. Esta sea apuntada por un puntero y sumarle 3 a través del puntero. Luego imprimir el resultado.

Solución: Esta es una posible solución:

```
#include <stdio.h>

int main()
     {
     int a;
     int *b;

     a = 5;
     b = &a;
     *b += 3;
     printf( "El valor de a es = %i\n", a );
     }
```

También se podía haber hecho: printf("El valor de a es = %i\n", *b);

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Arrays (Matrices)

Contenido del Capítulo:

¿Qué es un array?

> Nota: algunas personas conocen a los arrays como *arreglos, matrices o vectores.* Sin embargo, en este curso, vamos a usar el término array ya que es, según creo, el más extendido en la bibliografía sobre el tema.

La definición sería algo así:

> Un array es un conjunto de variables del mismo tipo que tienen el mismo nombre y se diferencian en el índice.

Pero ¿qué quiere decir esto y para qué lo queremos?. Pues bien, supongamos que somos un metereólogo y queremos guardar en el ordenador la temperatura que ha hecho cada hora del dia. Para darle cierta utilidad al final calcularemos la media de las temperaturas. Con lo que sabemos hasta ahora sería algo así (que nadie se moleste ni en probarlo):

```
#include <stdio.h>

int main()
     {
     /* Declaramos 24 variables, una para cada
hora del dia */
     int temp1,  temp2,  temp3,  temp4,  temp5,
temp6,  temp7,  temp8;
     int temp9,  temp10, temp11, temp12, temp13,
temp14, temp15, temp16;
```

```
     int temp17, temp18, temp19, temp20, temp21,
temp22, temp23, temp0;
     int media;

     /* Ahora tenemos que dar el valor de cada una
*/
     printf( "Temperatura de las 0: " );
     scanf( "%i", &temp0 );
     printf( "Temperatura de las 1: " );
     scanf( "%i", &temp1 );
     printf( "Temperatura de las 2: " );
     scanf( "%i", &temp2 );
     ...
     printf( "Temperatura de las 23: " );
     scanf( "%i", &temp23 );

     media = ( temp0 + temp1 + temp2 + temp3 +
temp4 + ... + temp23 ) / 24;
     printf( "\nLa temperatura media es %i\n",
media );
     }
```

NOTA: Los puntos suspensivos los he puesto para no tener que escribir todo y que no ocupe tanto, no se pueden usar en un programa.

Para acortar un poco el programa podríamos hacer algo así:

```
#include <stdio.h>

int main()
     {
     /* Declaramos 24 variables, una para cada
hora del dia */
     int temp1,  temp2,  temp3,  temp4,  temp5,
temp6,  temp7,  temp8;
     int temp9,  temp10, temp11, temp12, temp13,
temp14, temp15, temp16;
     int temp17, temp18, temp19, temp20, temp21,
temp22, temp23, temp0;
     int media;

     /* Ahora tenemos que dar el valor de cada una
*/
     printf( "Introduzca las temperaturas desde
las 0 hasta las 23 separadas por un espacion: " );
     scanf( "%i %i %i ... %i", &temp0, &temp1,
&temp2, ... &temp23 );

     media = ( temp0 + temp1 + temp2 + temp3 +
temp4 + ... + temp23 ) / 24;
     printf( "\nLa temperatura media es %i\n",
media );
     }
```

Lo que no deja de ser un peñazo. Y esto con un ejemplo que tiene tan sólo 24 variables, ¡¡imagínate si son más!!

Y precisamente aquí es donde nos vienen de perlas los arrays. Vamos a hacer el programa con un array. Usaremos nuestros conocimientos de bucles for y de scanf.

```
#include <stdio.h>

int main()
     {
     int temp[24]; /* Con esto ya tenemos
declaradas las 24 variables */
     float media = 0;
     int hora;

     /* Ahora tenemos que dar el valor de cada una
*/
     for( hora=0; hora<24; hora++ )
          {
          printf( "Temperatura de las %i: ", hora
);
          scanf( "%i", &temp[hora] );
          media += temp[hora];
          }
     media = media / 24;

     printf( "\nLa temperatura media es %f\n",
media );
     }
```

Como ves es un programa más rápido de escribir (y es menos aburrido hacerlo), y más cómodo para el usuario que el anterior.

Como ya hemos comentado cuando declaramos una variable lo que estamos haciendo es reservar una zona de la memoria para ella. Cuando declaramos un array lo que hacemos (en este ejemplo) es reservar espacio en memoria para 24 variables de tipo int. El tamaño del array (24) lo indicamos entre corchetes al definirlo. Esta es la parte de la definición que dice: *Un array es un conjunto de variables del mismo tipo que tienen el mismo nombre*.

La parte final de la definición dice: *y se diferencian en el índice*. En ejemplo recorremos la matriz mediante un bucle for y vamos dando valores a los distintos elementos de la matriz. Para indicar a qué elemento nos referimos usamos un número entre corchetes (en este caso la variable hora), este número es lo que se llama **Indice**. El primer elemento de la matriz en C tiene el índice 0, El segundo tiene el 1 y así sucesivamente. De modo que si queremos dar un valor al elemento 4 (índice 3) haremos:

```
    temp[ 3 ] = 20;
```

NOTA: No hay que confundirse. En la declaración del array el número entre corchetes es el número de elementos, en cambio cuando ya usamos la matriz el número entre corchetes es el índice.

[Arriba]

Declaración de un Array

La forma general de declarar un array es la siguiente:

```
    tipo_de_dato nombre_del_array[ dimensión ];
```

El *tipo_de_dato* es uno de los tipos de datos conocidos (int, char, float...) o de los definidos por el usuario con typdef. En el ejemplo era int.

El nombre_del_array es el nombre que damos al array, en el ejemplo era temp.

La dimensión es el número de elementos que tiene el array.

Como he indicado antes, al declarar un array reservamos en memoria tantas variables del *tipo_de_dato* como las indicada en *dimensión*.

[Arriba]

Sobre la dimensión de un Array

Hemos visto en el ejemplo que tenemos que indicar en varios sitios el tamaño del array: en la declaración, en el bucle for y al calcular la media. Este es un programa pequeño, en un programa mayor probablemente habrá que escribirlo muchas más veces. Si en un momento dado queremos cambiar la dimensión del array tendremos que cambiar todos. Si nos equivocamos al escribir el tamaño (ponemos 25 en vez de 24) cometeremos un error y puede que no nos demos cuenta. Por eso es mejor usar una constante con nombre, por ejemplo ELEMENTOS.

```
#include <stdio.h>

#define ELEMENTOS  24
int main()
     {
     int temp[ELEMENTOS]; /* Con esto ya tenemos
declaradas las 24 variables */
     float media = 0;
     int hora;

     /* Ahora tenemos que dar el valor de cada una
*/
     for( hora=0; hora<ELEMENTOS; hora++ )
          {
          printf( "Temperatura de las %i: ", hora
);
          scanf( "%i", &temp[hora] );
          media += temp[hora];
          }
     media = media / ELEMENTOS;

     printf( "\nLa temperatura media es %f\n",
media );
     }
Comprobado con DJGPP
```

Ahora con sólo cambiar el valor de elementos una vez lo estaremos haciendo en todo el programa.

[Arriba]

Inicializar un array

En C se pueden inicializar los arrays al declararlos igual que hacíamos con las variables. Recordemos que se podía hacer:

```
    int hojas = 34;
```

Pues con arrays se puede hacer:

```
    int temperaturas[24] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25, 24, 22,
                             21, 20, 18, 17, 16,
17, 15, 14, 14, 14, 13, 12 };
```

Ahora el elemento 0 (que será el primero), es decir temperaturas[0] valdrá 15. El elemento 1 (el segundo) valdrá 18 y así con todos. Vamos a ver un ejemplo:

```
#include <stdio.h>

int main()
     {
     int hora;
     int temperaturas[24] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25, 24,
                                  22, 21, 20, 18,
17, 16, 17, 15, 14, 14, 14, 13, 12 };

     for (hora=0 ; hora<24 ; hora++ )
          {
          printf( "La temperatura a las %i era de
%i grados.\n", hora, temperaturas[hora] );
          }
     }
Comprobado con DJGPP
```

Pero a ver quién es el habilidoso que no se equivoca al meter los datos, no es difícil olvidarse alguno. Hemos indicado al compilador que nos reserve memoria para un array de 24 elementos de tipo int. ¿Qué ocurre si metemos menos de los reservados? Pues no pasa nada, sólo que los elementos que falten valdrán cero.

```
#include <stdio.h>

int main()
     {
     int hora;
     /* Faltan los tres últimos elementos */
     int temperaturas[24] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25,
                           24, 22, 21, 20, 18, 17,
16, 17, 15, 14, 14 };

     for (hora=0 ; hora<24 ; hora++ )
          {
          printf( "La temperatura a las %i era de
%i grados.\n", hora, temperaturas[hora] );
          }
     }
Comprobado con DJGPP
```

El resultado será:

```
La temperatura a las 0 era de 15 grados.
```

```
La temperatura a las 1 era de 18 grados.
La temperatura a las 2 era de 20 grados.
La temperatura a las 3 era de 23 grados.
...
La temperatura a las 17 era de 17 grados.
La temperatura a las 18 era de 15 grados.
La temperatura a las 19 era de 14 grados.
La temperatura a las 20 era de 14 grados.
La temperatura a las 21 era de 0 grados.
La temperatura a las 22 era de 0 grados.
La temperatura a las 23 era de 0 grados.
```

Vemos que los últimos 3 elementos son nulos, que son aquellos a los que no hemos dado valores. El compilador no nos avisa que hemos metido menos datos de los reservados.

NOTA: Fíjate que para recorrer del elemento 0 al 23 (24 elementos) hacemos: for(hora=0; hora<24; hora++). La condición es que hora sea menos de 24. También podíamos haber hecho que hora!=24.

Ahora vamos a ver el caso contrario, metemos más datos de los reservados. Vamos a meter 25 en vez de 24. Si hacemos esto dependiendo del compilador obtendremos un error o al menos un warning (aviso). En unos compiladores el programa se creará y en otros no, pero aún así nos avisa del fallo. Debe indicarse que estamos intentando guardar un dato de más, no hemos reservado memoria para él.

Si la matriz debe tener una longitud determinada usamos este método de definir el número de elementos. En nuestro caso era conveniente, porque los dias siempre tienen 24 horas. Es conveniente definir el tamaño de la matriz para que nos avise si metemos más elementos de los necesarios.

En los demás casos podemos usar un método alternativo, dejar al ordenador que cuente los elementos que hemos metido y nos reserve espacio para ellos:

```
#include <stdio.h>

int main()
     {
     int hora;
     /* Faltan los tres últimos elementos */
     int temperaturas[] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25, 24,
                          22, 21, 20, 18, 17, 16,
17, 15, 14, 14 };

     for ( hora=0 ; hora<24 ; hora++ )
         {
         printf( "La temperatura a las %i era de
%i grados.\n", hora, temperaturas[hora] );
```

```
          }
     }
Comprobado con DJGPP
```

Vemos que no hemos especificado la dimensión del array temperaturas. Hemos dejado los corchetes en blanco. El ordenador contará los elementos que hemos puesto entre llaves y reservará espacio para ellos. De esta forma siempre habrá el espacio necesario, ni más ni menos. La pega es que si ponemos más de los que queríamos no nos daremos cuenta.

Para saber en este caso cuantos elementos tiene la matriz podemos usar el operador sizeof. Dividimos el tamaño de la matriz entre el tamaño de sus elementos y tenemos el número de elementos.

```
#include <stdio.h>

int main()
     {
     int hora;
     int elementos;
     int temperaturas[] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25,
                    24, 22, 21, 20, 18, 17, 16,
17, 15, 14, 14 };

     elementos = sizeof temperaturas /
sizeof(int);
     for ( hora=0 ; hora<elementos ; hora++ )
          {
          printf( "La temperatura a las %i era de
%i grados.\n", hora, temperas[hora] );
          }
     printf( "Han sido %i elementos.\n" ,
elementos );
     }
Comprobado con DJGPP
```

[Arriba]

Recorrer un array

En el apartado anterior veíamos un ejemplo que mostraba todos los datos de un array. Veíamos también lo que pasaba si metíamos más o menos elementos al inicializar la matriz. Ahora vamos a ver qué pasa si intentamos imprimir más elementos de los que hay en la matriz, en este caso intentamos imprimir 28 elementos cuando sólo hay 24:

```
#include <stdio.h>

int main()
     {
     int hora;
     /* Faltan los tres últimos elementos */
     int temperaturas[24] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25, 24,
                              22, 21, 20, 18, 17,
16, 17, 15, 14, 14, 13, 13, 12 };

     for (hora=0 ; hora<28 ; hora++ )
          {
          printf( "La temperatura a las %i era de
%i grados.\n", hora, temperaturas[hora] );
          }
     }
```

Comprobado con DJGPP

Lo que se obtiene en mi ordenador es:

```
La temperatura a las 22 era de 15 grados.
...
La temperatura a las 23 era de 12 grados.
La temperatura a las 24 era de 24 grados.
La temperatura a las 25 era de 3424248 grados.
La temperatura a las 26 era de 7042 grados.
La temperatura a las 27 era de 1 grados.
```

Vemos que a partir del elemento 24 (incluído) tenemos resultados extraños. Esto es porque nos hemos salido de los límites del array e intenta acceder al elemento temperaturas[25] y sucesivos que no existen. Así que nos muestra el contenido de la memoria que está justo detrás de temperaturas[23] (esto es lo más probable) que puede ser cualquiera. Al contrario que otros lenguajes C no comprueba los límites de los array, nos deja saltárnoslos a la torera. Este programa no da error al compilar ni al ejecutar, tan sólo devuelve resultados extraños. Tampoco bloqueará el sistema porque no estamos escribiendo en la memoria sino leyendo de ella.

Otra cosa muy diferente es meter datos en elementos que no existen. Veamos un ejemplo **(ni se te ocurra ejecutarlo)**:

```
#include <stdio.h>

int main()
     {
     int temp[24];
```

```
     float media = 0;
     int hora;

     for( hora=0; hora<28; hora++ )
          {
          printf( "Temperatura de las %i: ", hora
);
          scanf( "%i", &temp[hora] );
          media += temp[hora];
          }
     media = media / 24;

     printf( "\nLa temperatura media es %f\n",
media );
     }
```

Lo que sospechaba, lo he probado en mi ordenador y se ha bloqueado. He tenido que apagarlo. El problema ahora es que estamos intentando escribir en el elemento temp[24] que no existe y puede ser un lugar cualquiera de la memoria. Como consecuencia de esto podemos estar cambiando algún programa o dato de la memoria que no debemos y el sistema hace pluf. Así que mucho cuidado con esto.

[Arriba]

Punteros a arrays

Aquí tenemos otro de los importantes usos de los punteros, los punteros a arrays. Estos están íntimamente relacionados.

Para que un puntero apunte a un array se puede hacer de dos formas, una es apuntando al primer elemento del array:

```
    int *puntero;
    int temperaturas[24];

    puntero = &temperaturas[0];
```

El puntero apunta a la dirección del primer elemento. Otra forma equivalente, pero mucho más usada es:

```
    puntero = temperaturas;
```

Con esto también apuntamos al primer elemento del array. Fijaos que el puntero tiene que ser del mismo tipo que el array (en este caso int).

Ahora vamos a ver cómo acceder al resto de los elementos. Para ello empezamos por cómo funciona un array: Un array se guarda en posiciones seguidas en memoria, de tal forma que el segundo elemento va inmediatamente después del primero en la memoria. En un ordenador en el que el tamaño del tipo int es de 32 bits (4 bytes) cada elemento del array ocupará 4 bytes. Veamos un ejemplo:

```
#include <stdio.h>

int main()
     {
     int i;
     int temp[24];

     for( i=0; i<24; i++ )
          {
          printf( "La dirección del elemento %i es
%p.\n", i, &temp[i] );
          }
     }
```

NOTA: Recuerda que %p sirve para imprimir en pantalla la dirección de una variable en hexadecimal.

El resultado es (en mi ordenador):

```
La dirección del elemento 0 es 4c430.
La dirección del elemento 1 es 4c434.
La dirección del elemento 2 es 4c438.
La dirección del elemento 3 es 4c43c.
...
La dirección del elemento 21 es 4c484.
La dirección del elemento 22 es 4c488.
La dirección del elemento 23 es 4c48c.
```

(Las direcciones están en hexadecimal). Vemos aquí que efectivamente ocupan posiciones consecutivas y que cada una ocupa 4 bytes. Si lo representamos en una tabla:

4C430	4C434	4C438	4C43C
temp[0]	temp[1]	temp[2]	temp[3]

Ya hemos visto cómo funcionas los arrays por dentro, ahora vamos a verlo con punteros. Voy a poner un ejemplo:

```
#include <stdio.h>

int main()
     {
     int i;
     int temp[24];
     int *punt;

     punt = temp;

     for( i=0; i<24; i++ )
          {
          printf( "La dirección de temp[%i] es %p
y la de punt es %p.\n",
                  i, &temp[i], punt );
          punt++;
          }
     }
Comprobado con DJGPP
```

Cuyo resultado es:

```
La dirección de temp[0] es 4c430 y la de punt es
4c430.
La dirección de temp[1] es 4c434 y la de punt es
4c434.
La dirección de temp[2] es 4c438 y la de punt es
4c438.
...
La dirección de temp[21] es 4c484 y la de punt es
4c484.
La dirección de temp[22] es 4c488 y la de punt es
4c488.
La dirección de temp[23] es 4c48c y la de punt es
4c48c.
```

En este ejemplo hay dos líneas importantes (en negrita). La primera es *punt = temp*. Con esta hacemos que el punt apunte al primer elemento de la matriz. Si no hacemos esto punt apunta a un sitio cualquiera de la memoria y debemos recordar que no es conveniente dejar los punteros así, puede ser desastroso.

La segunda línea importante es *punt++*. Con esto incrementamos el valor de punt, pero curiosamente aunque incrementamos una unidad (punt++ equivale a punt=punt+1) el valor aumenta en 4. Aquí se muestra una de las características especiales de los punteros. Recordemos que en un puntero se guarda una dirección. También sabemos que un puntero apunta a un tipo de datos determinado (en este caso int). Cuando sumamos 1 a un puntero sumamos el tamaño del tipo al que apunta. En el ejemplo el puntero apunta a una variable de tipo int que es de 4 bytes, entonces al sumar 1 lo que hacemos es sumar 4 bytes.

Con esto lo que se consigue es apuntar a la siguiente posición int de la memoria, en este caso es el siguiente elemento de la matriz.

Operación	Equivalente	Valor de punt
punt = temp;	punt = &temp[0];	4c430
punt++;	sumar 4 al contenido de punt (4c430+4)	4c434
punt++;	sumar 4 al contenido de punt (4c434+4)	4c438

Cuando hemos acabado estamos en temp[24] que no existe. Si queremos recuperar el elemento 1 podemos hacer *punt = temp* otra vez o restar 24 a punt:

```
    punt -= 24;
```

con esto hemos restado 24 posiciones a punt (24 posiciones int*4 bytes por cada int= 96 posiciones).

Vamos a ver ahora un ejemplo de cómo recorrer la matriz entera con punteros y cómo imprimirla:

```
#include <stdio.h>

int main(int argc,char *argv[])
     {
     int temperaturas[24] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25, 24,
                                 22, 21, 20, 18,
17, 16, 17, 15, 14, 14, 13, 12, 12 };

     int *punt;
     int i;

     punt = temperaturas;

     for( i=0 ; i<24; i++ )
          {
          printf( "Elemento %i: %i\n", i, *punt );
          punt++;
          }
     }
Comprobado con DJGPP
```

Cuando acabamos punt apunta a temperaturas[24], y no al primer elemento, si queremos volver a recorrer la matriz debemos volver como antes al comienzo. Para evitar perder la referencia al primer elemento de la matriz (temperaturas[0]) se puede usar otra forma de recorrer la matriz con punteros:

```
#include <stdio.h>

int main(int argc,char *argv[])
     {
     int temperaturas[24] = { 15, 18, 20, 23, 22,
24, 22, 25, 26, 25, 24,
                                   22, 21, 20, 18,
17, 16, 17, 15, 14, 14, 13, 12, 12 };

     int *punt;
     int i;

     punt = temperaturas;

     for( i=0 ; i<24; i++ )
          {
          printf( "Elemento %i: %i\n", i,
*(punt+i) );
          }
     }
Comprobado con DJGPP
```

Con `*(punt+i)` lo que hacemos es tomar la dirección a la que apunta *punt* (la dirección del primer elemento de la matriz) y le sumamos *i* posiciones. De esta forma tenemos la dirección del elemento *i*. No estamos sumando un valor a *punt*, para sumarle un valor habría que hacer `punt++` o `punt+=algo`, así que *punt* siempre apunta al principio de la matriz.

Se podría hacer este programa sin usar *punt*. Sustituyendolo por *temperaturas* y dejar `*(temperaturas+i)`. Lo que no se puede hacer es: `temperaturas++;`.

Importante: Como final debo comentar que el uso de índices es una forma de maquillar el uso de punteros. El ordenador convierte los índices a punteros. Cuando al ordenador le decimos temp[5] en realidad le estamos diciendo *(temp+5). Así que usar índices es equivalente a usar punteros de una forma más cómoda.

[Arriba]

Paso de un array a una función

En C no podemos pasar un array entero a una función. Lo que tenemos que hacer es pasar un puntero al array. Con este puntero podemos recorrer el array:

```
#include <stdio.h>

int sumar( int *m )
     {
     int suma, i;

     suma = 0;
     for( i=0; i<10; i++ )
          {
          suma += m[i];
          }
     return suma;
     }

int main()
     {
     int contador;
     int matriz[10] = { 10, 11, 13, 10, 14, 9, 10,
18, 10, 10 };

     for( contador=0; contador<10; contador++ )
          printf( "   %3i\n", matriz[contador] );
     printf( "+ -----\n" );
     printf( "   %3i", sumar( matriz ) );
     }
Comprobado con DJGPP
```

Este programa tiene alguna cosilla adicional como que muestra toda la matriz en una columna. Además se usa para imprimir los números **%3i**. El 3 indica que se tienen que alinear los números a la derecha, así queda más elegante.

Como he indicado no se pasa el array, sino un puntero a ese array. Si probamos el truquillo de más arriba de usar sizeof para calcular el número de elementos no funcionará aquí. Dentro de la función suma añadimos la línea:

```
    printf( "Tamaño del array: %i Kb, %i bits\n",
sizeof m, (sizeof m)*8 );
```

Devolverá 4 (depende del compilador) que serán los bytes que ocupa el int (m es un puntero a int). ¿Cómo sabemos entonces cual es el tamaño del array dentro de la función? En este caso lo hemos puesto nosotros mismos, 10. Pero creo que lo mejor es utilizar constantes como en el apartado **Sobre la dimensión de un array.**

He dicho que el array no se puede pasar a una función, que se debe usar un puntero. Pero vemos que luego estamos usando m[i], esto lo podemos hacer porque como se ha mencionado antes el uso de índices en una forma que nos ofrece **C** de manejar punteros con matrices. Ya se ha visto que m[i] es equivalente a *(m+i).

Otras declaraciones equivalentes serían:

```
    int sumar( int m[] )
ó
    int sumar( int m[10] )
```

En realidad esta última no se suele usar, no es necesario indicar el número de elementos a la matriz.

int m[] e int *m son equivalentes.

[Arriba]

Ejercicios

Ejercicio 1:

Solución:

[Arriba]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Arrays Multidimensionales

Contenido del Capítulo:

¿Qué es un array bidimensional?

Hemos visto en el capítulo anterior lo conveniente que es usar un array cuando tenemos que usar muchas variables del mismo "tipo". Pero cuando avanzamos un poco más en programación podemos encontrarnos que los arrays también se nos quedan cortos. Podemos verlo utilizando el ejemplo del capítulo anterior:

"Supongamos que ahora queremos almacenar las temperaturas de toda la semana. Según lo que aprendimos en el capítulo anterior podríamos usar un array unidimiensional por cada día de la semana. En cada uno de esos arrays podríamos almacenar las temperaturas de cada día."

Una posible solución sería ésta:

```
#include <stdio.h>



int main()

     {

     int temp_dia1[24];

     int temp_dia2[24];
```

```
     int temp_dia3[24];

     ...

     int temp_dia7[24];



     float media = 0;

     int hora;



     for( hora=0; hora<24; hora++ ) {

          printf( "Temperatura de las %i
el día 1: ", hora );

          scanf( "%i", &temp_dia1[hora] );

          printf( "Temperatura de las %i
el día 2: ", hora );

          scanf( "%i", &temp_dia1[hora] );

          printf( "Temperatura de las %i
el día 3: ", hora );

          scanf( "%i", &temp_dia1[hora] );

          ...

          printf( "Temperatura de las %i
el día 7: ", hora );

          scanf( "%i", &temp_dia1[hora] );
```

```
          media += temp_dia1[hora] +
temp_dia2[hora] + ... + temp_dia3[hora];

     }

     media = media / 24 / 7;



     printf( "\nLa temperatura media de
toda la semana es %f\n", media );

     return 0;

}
```

Fichero: cap10_ejemplo1.c

> Nota: os recuerdo que los puntos suspensivos los he puesto para ahorrar espacio, no pueden usarse en un programa.

Creo que está claro que nuevamente queda claro que el programa sería un engorro, sobre todo si queremos almacenar las temperaturas de, por ejemplo, un mes.

Para declarar un array bidimensional usaremos el siguiente formato:

```
tipo_de_dato nombre_del_array[ filas ] [ columnas ];
```

Como se puede apreciar, la declaración es igual que la de un array unidimensional al que le añadimos una nueva dimensión.

El programa anterior quedaría ahora así:

```
#include <stdio.h>

#define DIAS    7
```

```c
#define HORAS   24



int main()

     {

     int temp[DIAS][HORAS];



     float media = 0;

     int hora, dia;



     for( dia=0 ; dia<DIAS ; dia++ ) {

          for( hora=0 ; hora<HORAS ;
hora++ ) {

              printf( "Temperatura de las
%d el día %d: ", hora, dia );

              scanf( "%i", &temp[dia][hora]
);

              media += temp[dia][hora];

          }

     }

     media = media / HORAS / DIAS;
```

```
      printf( "\nLa temperatura media de
 toda la semana es %f\n", media );


      return 0;


 }
```

Fichero: cap10_ejemplo2.c

No tenemos por qué limitarnos a arrays bidimensionales, podemos usar cuantas dimensiones queramos. Para un array de tres dimensiones podríamos hacer:

```
int temp[MESES][DIAS][HORAS];
```

[Arriba]

Inicializar un array multidimiensional

Vimos en el anterior capítulo que a los arrays unidimensionales se les podían dar valores iniciales:

int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25, 24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 14, 13, 12 };

Con los array multidimensionales también se puede, pero hay que agrupar entre {} cada fila. Por ejemplo:

```
int temperaturas[3][5] = {
               { 15, 17, 20, 25, 10 },
               { 18, 20, 21, 23, 18 },
               { 12, 17, 23, 29, 16 } };
```

El formato a utilizar sería el siguiente:

```
int variable[ filas ][ columnas ] = {
            { columnas de la fila 1 },
            { columnas de la fila 2 },
            ... ,
            { columnas de la última fila },
            };
```

No debemos olvidar el ';' al final.

[Arriba]

Ejercicios

Ejercicio 1:

Solución:

[Arriba]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Strings (Cadenas de texto)

Contenido del Capítulo:

Introducción

Vamos a ver por fin cómo manejar texto con ***C***, hasta ahora sólo sabíamos cómo mostrarlo por pantalla.

Para empezar diré que en ***C*** no existe un tipo string como en otros lenguajes. No existe un tipo de datos para almacenar texto, se utilizan arrays de chars. Funcionan igual que los demás arrays con la diferencia que ahora jugamos con letras en vez de con números.

Se les llama cadenas, strings o tiras de caracteres. A partir de ahora les llamaremos **cadenas**.

Para declarar una cadena se hace como un array:

```
    char texto[20];
```

Al igual que en los arrays no podemos meter más de 20 elementos en la cadena. Vamos a ver un ejemplo para mostrar el nombre del usuario en pantalla:

```
#include <stdio.h>

int main()
     {
     char nombre[20];

     printf( "Introduzca su nombre (20 letras
máximo): " );
     scanf( "%s", nombre );
     printf( "\nEl nombre que ha escrito es:
%s\n", nombre );
     }
```
Comprobado con DJGPP

Vemos cosas curiosas como por ejemplo que en el scanf no se usa el símbolo **&**. No hace falta porque es un array, y ya sabemos que escribir el nombre del array es equivalente a poner &nombre[0].

También puede llamar la atención la forma de imprimir el array. Con sólo usar %s ya se imprime todo el array. Ya veremos esto más adelante.

Si alguno viene de algún otro lenguaje esto es importante: en ***C*** no se puede hacer esto:

```
int main()
     {
     char texto[20];

     texto = "Hola";
     }
```
Comprobado con DJGPP

[Arriba]

Las cadenas por dentro

Es interesante saber cómo funciona una cadena por dentro, por eso vamos a ver primero cómo se inicializa una cadena.

```
#include <stdio.h>
```

```
int main()
     {
     char nombre[] = "Gorka";

     printf( "Texto: %s\n", nombre );
     printf( "Tamaño de la cadena: %i bytes\n",
sizeof nombre );
     }
```
Comprobado con DJGPP

Resultado al ejecutar:

```
Texto: Gorka
Tamaño de la cadena: 6 bytes
```

¡Qué curioso! La cadena es "Gorka", sin embargo nos dice que ocupa 6 bytes. Como cada elemento (char) ocupa un byte eso quiere decir que la cadena tiene 6 elementos. ¡Pero si "Gorka" sólo tiene 5! ¿Por qué? Muy sencillo, porque al final de una cadena se pone un símbolo '**\0**' que significa "Fin de cadena". De esta forma cuando queremos escribir la cadena basta con usar %s y el programa ya sabe cuántos elementos tiene que imprimir, hasta que encuentre '\0'.

El programa anterior sería equivalente a:

```
#include <stdio.h>

int main(int argc,char *argv[])
     {
     char nombre[] = { 'G', 'o', 'r', 'k', 'a',
'\0' };

     printf( "Texto: %s\n", nombre );
     }
```
Comprobado con DJGPP

Aquí ya se ve que tenemos 6 elementos. Pero, ¿Qué pasaría si no pusiéramos '\0' al final?

```
#include <stdio.h>

int main()
     {
     char nombre[] = { 'G', 'o', 'r', 'k', 'a' };

     printf( "Texto: %s\n", nombre );
     }
```
Comprobado con DJGPP

En mi ordenador salía:

```
Texto: Gorka-
Tamaño de la cadena: 5 bytes
```

Pero en el tuyo después de "Gorka" puede aparecer cualquier cosa. Lo que aquí sucede es que no encuentra el símbolo '\0' y no sabe cuándo dejar de imprimir. Afortunadamente, cuando metemos una cadena se hace de la primera forma y el ***C*** se encarga de poner el dichoso símbolo al final.

Es importante no olvidar que la longitud de una cadena es la longitud del texto más el símbolo de fin de cadena. Por eso cuando definamos una cadena tenemos que reservarle un espacio adicional. Por ejemplo:

```
    char nombre[6] = "Gorka";
```

Si olvidamos esto podemos tener problemas.

[Arriba]

Funciones de manejo de cadenas

Existen unas cuantas funciones el la biblioteca estándar de ***C*** para el manejo de cadenas:

- **strlen**
- **strcpy**
- **strcat**
- **sprintf**
- **strcmp**

Para usar estas funciones hay que añadir la directiva:

```
    #include <string.h>
```

strlen

Esta función nos devuelve el número de caracteres que tiene la cadena (sin contar el '\0').

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char texto[]="Gorka";
     int longitud;

     longitud = strlen(texto);
     printf( "La cadena \"%s\" tiene %i
caracteres.\n", texto, longitud );
     }
```

Comprobado con DJGPP

Crea tus propias funciones: Vamos a ver cómo se haría esta función si no dispusiéramos de ella. Si no te enteras cómo funciona consulta **Recorrer cadenas con punteros.**

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char texto[]="Gorka";
     char *p;
     int longitud=0;

     p = texto;
     while (*p!='\0')
          {
          longitud++;
          printf( "%c\n", *p ); /* Mostramos la
letra actual */
          p++;                  /* Vamos a la
siguiente letra */
          }
     printf( "La cadena \"%s\" tiene %i
caracteres.\n", texto, longitud );
     }
```

Comprobado con DJGPP

Para medir la longitud de la cadena usamos un puntero para recorrerla (el puntero *p*). Hacemos que *p* apunte a *texto*. Luego entramos en un bucle while. La condición del bucle comprueba si se ha llegado al fin de cadena ('\0'). Si no es así suma 1 a *longitud*, muestra la letra por pantalla e incrementa el puntero en 1 (con esto pasamos a la siguiente letra).

strcpy

```
     #include <string.h>

     char *strcpy(char *cadena1, const char
*cadena2);
```

Copia el contenido de *cadena2* en *cadena1*. *cadena2* puede ser una variable o una cadena directa (por ejemplo "hola"). Debemos tener cuidado de que la cadena destino (cadena1) tenga espacio suficiente para albergar a la cadena origen (cadena2).

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char textocurso[] = "Este es un curso de C.";
     char destino[50];

     strcpy( destino, textocurso );
     printf( "Valor final: %s\n", destino );
     }
```

Comprobado con DJGPP

Vamos a ver otro ejemplo en el que la cadena destino es una cadena constante ("Este es un curso de C") y no una variable. Además en este ejemplo vemos que la cadena origen es sustituida por la cadena destino totalmete. Si la cadena origen es más larga que la destino, se eliminan las letras adicionales.

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char destino[50] = "Esto no es un curso de
HTML sino un curso de C.";

     printf( "%s\n", destino );
     strcpy( destino, "Este es un curso de C." );
     printf( "%s\n", destino );
     }
```

Comprobado con DJGPP

strcat

```
     #include <string.h>

     char *strcat(char *cadena1, const char
*cadena2);
```

Copia la *cadena2* al final de la *cadena1*.

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char nombre_completo[50];
     char nombre[]="Gorka";
     char apellido[]="Urrutia";

     strcpy( nombre_completo, nombre );
     strcat( nombre_completo, " " );
     strcat( nombre_completo, apellido );
     printf( "El nombre completo es: %s.\n",
nombre_completo );
     }
```

Comprobado con DJGPP

Como siempre tenemos que asegurarnos que la variable en la que metemos las demás cadenas tenga el tamaño suficiente. Con la primera línea metemos el nombre en *nombre_completo*. Usamos strcpy para asegurarnos de que queda borrado cualquier dato anterior. Luego usamos un strcat para añadir un espacio y finalmente metemos el apellido.

sprintf

```
     #include <stdio.h>

     int sprintf(char *destino, const char
*format, ...);
```

Funciona de manera similar a printf, pero en vez de mostrar el texto en la pantalla lo guarda en una variable (*destino*). El valor que devuelve (int) es el número de caracteres guardados en la variable *destino*.

Con sprintf podemos repetir el ejemplo de strcat de manera más sencilla:

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char nombre_completo[50];
     char nombre[]="Gorka";
     char apellido[]="Urrutia";

     sprintf( nombre_completo, "%s %s", nombre,
apellido );
     printf( "El nombre completo es: %s.\n",
nombre_completo );
     }
```
Comprobado con DJGPP

Se puede aplicar a sprintf todo lo que valía para printf.

[Arriba]

strcmp

```
     #include <string.h>

     int strcmp(const char *cadena1, const char
*cadena2);
```

Compara *cadena1* y *cadena2*. Si son iguales devuelve 0. Un número negativo si *cadena1* va antes que *cadena2* y un número positivo si es al revés:

- `cadena1 == cadena2 -> 0`
- `cadena1 > cadena2 -> número negativo`
- `cadena1 < cadena2 -> número positivo`

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char nombre1[]="Gorka";
     char nombre2[]="Pedro";

     printf( "%i", strcmp(nombre1,nombre2));
     }
```
Comprobado con DJGPP

[Arriba]

toupper() y tolower() - Convertir a mayúsculas y minúsculas

La función tolower() nos permite convertir una cadena a minúsculas:

```
char *tolower( char *cadena );
```

toupper cumple la función contraria, convierte la cadena a mayúsculas:

```
char *toupper( char *cadena );
```

[Arriba]

Entrada de cadenas por teclado (scanf y gets)

- **scanf**
- **gets**
- **Qué son los buffer y cómo funcionan**
- **getchar**

scanf

Hemos visto en capítulos anteriores el uso de scanf para números, ahora es el momento de ver su uso con cadenas.

Scanf almacena en memoria (en un buffer) lo que vamos escribiendo. Cuando pulsamos ENTER (o Intro o Return, como se llame en cada teclado) lo analiza, comprueba si el formato es correcto y por último lo mete en la variable que le indicamos.

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char cadena[30];

     printf( "Escribe una palabra: " );
     fflush( stdout );
     scanf( "%s", cadena );
     printf( "He guardado: \"%s\" \n", cadena );
     }
```

Comprobado con DJGPP

Ejecutamos el programa e introducimos la palabra "hola". Esto es lo que tenemos:

```
Escribe una palabra: hola
He guardado: "hola"
```

Si ahora introducimos "hola amigos" esto es lo que tenemos:

```
Escribe una palabra: hola amigos
He guardado: "hola"
```

Sólo nos ha cogido la palabra "hola" y se ha olvidado de amigos. ¿Por qué? pues porque scanf toma una palabra como cadena. Usa los espacios para separar variables.

Es importante siempre asegurarse de que no vamos a almacenar en *cadena* más letras de las que caben. Para ello debemos limitar el número de letras que le va a introducir scanf. Si por ejemplo queremos un máximo de 5 caracteres usaremos %5s:

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char cadena[6];

     printf( "Escribe una palabra: " );
     fflush( stdout );
     scanf( "%5s", cadena );
     printf( "He guardado: \"%s\" \n", cadena );
     }
```

Comprobado con DJGPP

Si metemos una palabra de 5 letras (no se cuenta '\0') o menos la recoge sin problemas y la guarda en *cadena*.

```
Escribe una palabra: Gorka
He guardado: "Gorka"
```

Si metemos más de 5 letras nos cortará la palabra y nos dejará sólo 5.

```
Escribe una palabra: Juanjo
He guardado: "Juanj"
```

Scanf tiene más posibilidades (consulta la ayuda de tu compilador), entre otras permite controlar qué caracteres entramos. Supongamos que sólo queremos coger las letras mayúsculas:

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char cadena[30];

     printf( "Escribe una palabra: " );
     fflush( stdout );
     scanf( "%[A-Z]s", cadena );
     printf( "He guardado: \"%s\" \n", cadena );
     }
```

Comprobado con DJGPP

Guarda las letras mayúsculas en la variable hasta que encuentra una minúscula:

```
Escribe una palabra: Hola
He guardado: "H"
```

```
Escribe una palabra: HOLA
He guardado: "HOLA"
```

```
Escribe una palabra: AMigOS
He guardado: "AM"
```

[Arriba]

gets

Esta función nos permite introducir frases enteras, incluyendo espacios.

```
     #include <stdio.h>
```

```
     char *gets(char *buffer);
```

Almacena lo que vamos tecleando en la variable *buffer* hasta que pulsamos ENTER. Si se ha almacenado algún caracter en *buffer* le añade un '\0' al final y devuelve un puntero a su dirección. Si no se ha almacenado ninguno devuelve un puntero NULL.

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char cadena[30];
     char *p;

     printf( "Escribe una palabra: " );
     fflush( stdout );
     p = gets( cadena );
     if (p) printf( "He guardado: \"%s\" \n",
cadena );
     else printf( "No he guardado nada!\n" );
     }
```
Comprobado con DJGPP

Esta función es un poco peligrosa porque no comprueba si nos hemos pasado del espacio reservado (de 29 caracteres en este ejemplo: 29letras+'\0').

[Arriba]

Qué son los buffer y cómo funcionan

```
#include <stdio.h>

int main()
     {
     char ch;
     char nombre[20], apellido[20], telefono[10];
     printf( "Escribe tu nombre: " );
     scanf( "%[A-Z]s", nombre );
     printf( "Lo que recogemos del scanf es:
%s\n", nombre );
     printf( "Lo que había quedado en el buffer: "
);
     while( (ch=getchar())!='\n' )
            printf( "%c", ch );
     }
```
Comprobado con DJGPP

```
Escribe tu nombre: GORka
Lo que recogemos del scanf es: GOR
Lo que había quedado en el buffer: ka
```

Cuidado con scanf!!!

```
#include <stdio.h>

int main()
     {
     char nombre[20], apellido[20], telefono[10];
     printf( "Escribe tu nombre: " );
     scanf( "%s", nombre );
     printf( "Escribe tu apellido: " );
     gets( apellido );
     }
```

Comprobado con DJGPP

[Arriba]

getchar

[Arriba]

Recorrer cadenas con punteros

Las cadenas se pueden recorrer de igual forma que hacíamos con las matrices, usando punteros.

Vamos a ver un ejemplo: Este sencillo programa cuenta los espacios y las letras 'e' que hay en una cadena.

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char cadena[]="Gorka es un tipo estupendo";
     char *p;
     int espacios=0, letras_e=0;

     p = cadena;
```

```
     while (*p!='\0')
           {
           if (*p==' ') espacios++;
           if (*p=='e') letras_e++;
           p++;
           }
     printf( "En la cadena \"%s\" hay:\n", cadena
);
     printf( "  %i espacios\n", espacios );
     printf( "  %i letras e\n", letras_e );
     }
```
Comprobado con DJGPP

Para recorrer la cadena necesitamos un puntero *p* que sea de tipo char. Debemos hacer que *p* apunte a la cadena (p=cadena). Así *p* apunta a la dirección del primer elemento de la cadena. El valor de **p* sería por tanto 'G'. Comenzamos el bucle. La condición comprueba que no se ha llegado al final de la cadena (*p!='\0'), recordemos que '\0' es quien marca el final de ésta. Entonces comprobamos si en la dirección a la que apunta *p* hay un espacio o una letra e. Si es así incrementamos las variables correspondientes. Una vez comprobado esto pasamos a la siguiente letra (*p++*).

Dos cosas muy importantes: primero no debemos olvidarnos nunca de inicializar un puntero, en este caso hacer que apunte a *cadena*. Segundo no debemos olvidarnos de incrementar el puntero dentro del bucle (p++), sino estaríamos en un bucle infinito siempre comprobando el primer elemento.

En la condición del bucle podíamos usar símplemente: while (!*p), que es equivalente a (*p!='\0').

En este otro ejemplo sustituímos los espacios por guiones:

```
#include <stdio.h>
#include <string.h>

int main()
     {
     char cadena[]="Gorka es un tipo estupendo";
     char *p;

     p = cadena;
     while (*p!='\0')
           {
           if (*p==' ') *p = '-';
           p++;
           }
     printf( "La cadena queda: \"%s\" \n", cadena
);
     }
```
Comprobado con DJGPP

y se obtiene:

```
La cadena queda: "Gorka-es-un-tipo-estupendo"
```

[Arriba]

Arrays de cadenas

Un array de cadenas puede servirnos para agrupar una serie de mensajes. Por ejemplo todos los mensajes de error de un programa. Luego para acceder a cada mensaje basta con usar su número.

```
#include <stdio.h>
#include <string.h>

int error( int errnum )
    {
    char *errores[] = {
         "No se ha producido ningún error",
         "No hay suficiente memoria",
         "No hay espacio en disco",
         "Me he cansado de trabajar"
         };

    printf( "Error número %i: %s.\n", errnum,
errores[errnum] );
    exit( -1 );
    }

int main()
     {
     error( 2 );
     }
```

Comprobado con DJGPP

El resultado será:

```
Error número 2: No hay espacio en disco.
```

Un array de cadenas es en realidad un array de punteros a cadenas. El primer elemento de la cadena ("No se ha producido ningún error") tiene un espacio reservado en memoria y *errores[0]* apunta a ese espacio.

[Arriba]

Ordenar un array de cadenas

Vamos a ver un sencillo ejemplo de ordenación de cadenas. En el ejemplo tenemos que ordenar una serie de dichos populares:

```
#include <stdio.h>
#include <string.h>

#define ELEMENTOS       5

int main()
     {
     char *dichos[ELEMENTOS] = {
          "La avaricia rompe el saco",
          "Más Vale pájaro en mano que ciento
volando",
          "No por mucho madrugar amanece más
temprano",
          "Año de nieves, año de bienes",
          "A caballo regalado no le mires el
diente"
          };
     char *temp;
     int i, j;

     printf( "Lista desordenada:\n" );
     for( i=0; i<ELEMENTOS; i++ )
          printf( "  %s.\n", dichos[i] );
     for( i=0; i<ELEMENTOS-1; i++ )
          for( j=0; j<ELEMENTOS; j++ )
               if (strcmp(dichos[i], dichos[j])>0)
                  {
                  temp = dichos[i];
                  dichos[i] = dichos[j];
                  dichos[j] = temp;
                  }
     printf( "Lista ordenada:\n" );
     for( i=0; i<ELEMENTOS; i++ )
          printf( "  %s.\n", dichos[i] );
     }
```

Comprobado con DJGPP

Este método se conoce como el método de burbuja.

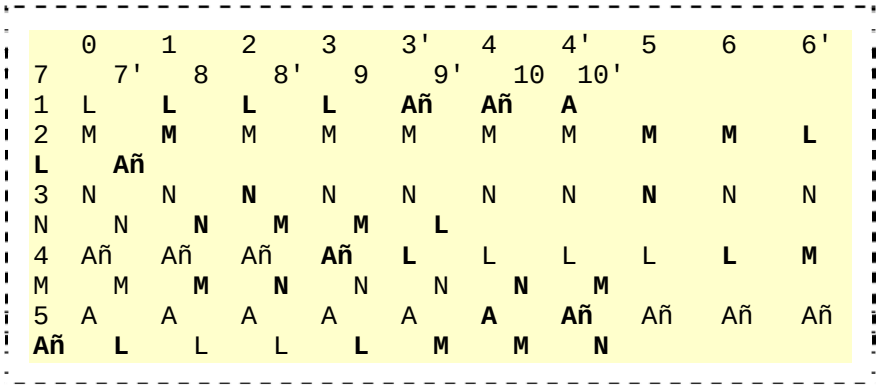
Cómo funciona el programa:

1.- Tomamos el primer elemento de la matriz. Lo comparamos con todos los siguientes. Si alguno es anterior los intercambiamos. Cuando

acabe esta primera vuelta tendremos "A caballo regalado no le mires el diente" en primera posición.

2.- Tomamos el segundo elemento. Lo comparamos con el tercero y siguientes. Si alguno es anterior los intercambiamos. Al final de esta vuelta quedará "A caballo regalado no le mires el diente" en segunda posición.

Para mayor claridad (eso espero) voy a sustituir cada cadena por su primera letra (menos la de "Año de nieves..." que la sustituyo por Añ). Y así represento el proceso:



	0	1	2	3	3'	4	4'	5	6	6'
7	7'	8	8'	9	9'	10	10'			
1	L	**L**	**L**	**L**	**Añ**	**Añ**	**A**			
2	M	**M**	M	M	M	M	M	**M**	**M**	**L**
L	**Añ**									
3	N	N	**N**	N	N	N	N	**N**	N	N
N	N	**N**	**M**	**M**	**L**					
4	Añ	Añ	Añ	**Añ**	**L**	L	L	L	**L**	**M**
M	M	**M**	**N**	N	N	**N**	**M**			
5	A	A	A	A	A	**A**	**Añ**	Añ	Añ	Añ
Añ	**L**	L	L	**L**	**M**	**M**	**N**			

[Arriba]

Ejercicios

Ejercicio 1: Crear un programa que tome una frase e imprima cada una de las palabras en una línea:

```
Introduzca una frase: La programación en C es divertida
Resultado:
La
programación
en
C
es
divertida
```

Solución:

```c
#include <stdio.h>

int main() {
    char frase[100];
    int i = 0;
```

```
    printf( "Escriba una frase: " );
    gets( frase );
    printf( "Resultado:\n" );
    while ( frase[i]!='\0' ) {
        if ( frase[i]==' ' )
                printf( "\n" );
        else
                printf( "%c", frase[i] );
        i++;
    }
    system( "pause" );
    return 0;
}
```

Ejercicio 2: Escribe un programa que después de introducir una palabra convierta alternativamente las letras a mayúsculas y minúsculas:

```
Introduce una palabra: chocolate
Resultado: ChOcoLaTe
```

Solución:

```
#include <stdio.h>

int main() {
    char palabra[100];
    int i = 0, j = 0;

    printf( "Escribe una palabra: " );
    gets( palabra );
    printf( "Resultado:\n" );
    while ( palabra[i]!='\0' ) {
        if ( j==0 )
                printf( "%c", toupper(
palabra[i] ) );
        else
                printf( "%c", tolower(
palabra[i] ) );
        j = 1 - j;
        i++;
    }
    printf( "\n" );
    system( "pause" );
    return 0;
}
```

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Funciones (Avanzado)

Contenido del Capítulo:

Introducción

Pasar argumentos a un programa

Ya sabemos cómo pasar argumentos a una función. La función **main** también acepta argumentos. Sin embargo sólo se le pueden pasar dos argumentos. Veamos cuáles son y cómo se declaran:

```
int main( int argc, char *argv[] )
```

El primer argumento es **argc** (**arg**ument **c**ount). Es de tipo int e indica el número de argumentos que se le han pasado al programa.

El segundo es **argv** (**arg**ument **v**alues). Es un array de strings (o puntero a puntero a char). En el se almacenan los parámetros. Normalmente (como siempre depende del compilador) el primer elemento (argv[0]) es el nombre del programa con su ruta. El segundo (argv[1]) es el primer parámetro, el tercero (argv[2]) el segundo parámetro y así hasta el final.

A los argumentos de main se les suele llamar siempre así, no es necesario pero es costumbre.

Veamos un ejemplo para mostrar todos los parámetros de un programa:

```
#include<stdio.h>

int main(int argc,char *argv[])
     {
     int i;
     for( i=0 ; i<argc ; i++ )
          printf( "Argumento %i: %s\n", i, argv[i]
);
     }
```
Comprobado con DJGPP

Si por ejemplo llamamos al programa `argumentos.c` y lo compilamos (`argumentos.exe`) podríamos teclear (lo que está en negrita es lo que tecleamos):

```
    c:\programas> argumentos hola amigos
```

Tendríamos como salida:

```
    Argumento 0: c:\programas\argumentos.exe
    Argumento 1: hola
    Argumento 2: amigos
```

Pero si en vez de eso tecleamos:

```
    c:\programas> argumentos "hola amigos"
```

Lo que tendremos será:

```
    Argumento 0: c:\programas\argumentos.exe
    Argumento 1: hola amigos
```

[Arriba]

Ejemplos enviados

Enviado por Angel:

```
#include <stdio.h>

int main(int argc,char *argv[])
        {
        int par1, par2;
        par1= *argv[1] - '0';
        par2= *argv[2] - '0';
        printf("%d + %d =
%d\n",par1,par2,par1+par2);
        }
```
Comprobado con DJGPP

Este programa espera dos argumentos. Ambos deben ser números del 0 al 9. El programa los coge y los suma.

[Arriba]

Seguridad

[Arriba]

Ejercicios

Ejercicio :

Solución:

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Estructuras

Contenido

Estructuras

Supongamos que queremos hacer una agenda con los números de teléfono de nuestros amigos. Necesitaríamos un array de Cadenas para almacenar sus nombres, otro para sus apellidos y otro para sus números de teléfono. Esto puede hacer que el programa quede desordenado y difícil de seguir. Y aquí es donde vienen en nuestro auxilio las estructuras.

Para definir una estructura usamos el siguiente formato:

```
struct nombre_de_la_estructura {
       campos de estructura;
       };
```

NOTA: Es importante no olvidar el ';' del final, si no a veces se obtienen errores extraños.

Para nuestro ejemplo podemos crear una estructura en la que almacenaremos los datos de cada persona. Vamos a crear una declaración de estructura llamada *amigo*:

```
struct estructura_amigo {
       char nombre[30];
       char apellido[40];
       char telefono[10];
       char edad;
```

```
        };
```

A cada elemento de esta estructura (nombre, apellido, teléfono) se le llama campo o **miembro**. (NOTA: He declarado *edad* como char porque no conozco a nadie con más de 127 años.

Ahora ya tenemos definida la estructura, pero aun no podemos usarla. Necesitamos declarar una variable con esa estructura.

```
struct estructura_amigo amigo;
```

Ahora la variable *amigo* es de tipo *estructura_amigo*. Para acceder al nombre de *amigo* usamos: **amigo.nombre**. Vamos a ver un ejemplo de aplicación de esta estructura. (NOTA: En el siguiente ejemplo los datos no se guardan en disco así que cuanda acaba la ejecución del programa se pierden).

```
#include <stdio.h>

struct estructura_amigo {  /* Definimos la
estructura estructura_amigo */
       char nombre[30];
       char apellido[40];
       char telefono[10];
       char edad;
       };

struct estructura_amigo amigo;

int main()
     {
     printf( "Escribe el nombre del amigo: " );
     fflush( stdout );
     scanf( "%s", &amigo.nombre );
     printf( "Escribe el apellido del amigo: " );
     fflush( stdout );
     scanf( "%s", &amigo.apellido );
     printf( "Escribe el número de teléfono del
amigo: " );
     fflush( stdout );
     scanf( "%s", &amigo.telefono );
     printf( "El amigo %s %s tiene el número:
%s.\n", amigo.nombre,
             amigo.apellido, amigo.telefono );
     }
```

Comprobado con DJGPP

Este ejemplo estaría mejor usando gets que scanf, ya que puede haber nombres compuestos que scanf no cogería por los espacios.

Se podría haber declarado directamente la variable *amigo*:

```
struct estructura_amigo {
       char nombre[30];
       char apellido[40];
       char telefono[10];
       } amigo;
```

[Arriba]

Arrays de estructuras

Supongamos ahora que queremos guardar la información de varios amigos. Con una variable de estructura sólo podemos guardar los datos de uno. Para manejar los datos de más gente (al conjunto de todos los datos de cada persona se les llama REGISTRO) necesitamos declarar arrays de estructuras.

¿Cómo se hace esto? Siguiendo nuestro ejemplo vamos a crear un array de *ELEMENTOS* elementos:

```
struct estructura_amigo amigo[ELEMENTOS];
```

Ahora necesitamos saber cómo acceder a cada elemento del array. La variable definida es *amigo*, por lo tanto para acceder al primer elemento usaremos *amigo[0]* y a su miembro *nombre*: *amigo[0].nombre*. Veamoslo en un ejemplo en el que se supone que tenemos que meter los datos de tres amigos:

```
#include <stdio.h>

#define ELEMENTOS       3

struct estructura_amigo {
       char nombre[30];
       char apellido[40];
       char telefono[10];
       int edad;
       };

struct estructura_amigo amigo[ELEMENTOS];

int main()
     {
     int num_amigo;
```

```
     for( num_amigo=0; num_amigo<ELEMENTOS;
num_amigo++ )
          {
          printf( "\nDatos del amigo número
%i:\n", num_amigo+1 );
          printf( "Nombre: " ); fflush( stdout );
          gets(amigo[num_amigo].nombre);
          printf( "Apellido: " ); fflush( stdout
);
          gets(amigo[num_amigo].apellido);
          printf( "Teléfono: " ); fflush( stdout
);
          gets(amigo[num_amigo].telefono);
          printf( "Edad: " ); fflush( stdout );
          scanf( "%i", &amigo[num_amigo].edad );
          while(getchar()!='\n');
          }
     /* Ahora imprimimos sus datos */
     for( num_amigo=0; num_amigo<ELEMENTOS;
num_amigo++ )
          {
          printf( "El amigo %s ",
amigo[num_amigo].nombre );
          printf( "%s tiene ",
amigo[num_amigo].apellido );
          printf( "%i años ",
amigo[num_amigo].edad );
          printf( "y su teléfono es el %s.\n" ,
amigo[num_amigo].telefono );
          }
     }
```
Comprobado con DJGPP

IMPORTANTE: Quizás alguien se pregunte qué pinta la línea esa de *while(getchar()!='\n');*. Esta línea se usa para vaciar el buffer de entrada. Para más información consulta **Qué son los buffer y cómo funcionan.**

[Arriba]

Inicializar una estructura

A las estructuras se les pueden dar valores iniciales de manera análoga a como hacíamos con los arrays. Primero tenemos que definir la estructura y luego cuando declaramos una variable como estructura le damos el valor inicial que queramos. Recordemos que esto no es en absoluto necesario. Para la estructura que hemos definido antes sería por ejemplo:

```
struct estructura_amigo amigo = {
       "Juanjo",
```

```
       "Lopez",
       "592-0483",
       30
       };
```

NOTA: En algunos compiladores es posible que se exiga poner antes de struct la palabra *static*.

Por supuesto hemos de meter en cada campo el tipo de datos correcto. La definición de la estructura es:

```
struct estructura_amigo {
        char nombre[30];
        char apellido[40];
        char telefono[10];
        int edad;
        };
```

por lo tanto el nombre ("Juanjo") debe ser una cadena de no más de 29 letras (recordemos que hay que reservar un espacio para el símbolo '\0'), el apellido ("Lopez") una cadena de menos de 39, el teléfono una de 9 y la edad debe ser de tipo char.

Vamos a ver la inicialización de estructuras en acción:

```
#include <stdio.h>

struct estructura_amigo {
        char nombre[30];
        char apellido[40];
        char telefono[10];
        int edad;
        };

struct estructura_amigo amigo = {
       "Juanjo",
       "Lopez",
       "592-0483",
       30
       };

int main()
      {
      printf( "%s tiene ", amigo.apellido );
      printf( "%i años ", amigo.edad );
      printf( "y su teléfono es el %s.\n" ,
amigo.telefono );
      }
```

Comprobado con DJGPP

También se puede inicializar un array de estructuras de la forma siguiente:

```
struct estructura_amigo amigo[] =
       {
       "Juanjo", "Lopez", "504-4342", 30,
       "Marcos", "Gamindez", "405-4823", 42,
       "Ana", "Martinez", "533-5694", 20
       };
```

En este ejemplo cada línea es un registro. Como sucedía en los arrays si damos valores iniciales al array de estructuras no hace falta indicar cuántos elementos va a tener. En este caso la matriz tiene 3 elementos, que son los que le hemos pasado.

[Arriba]

Punteros a estructuras

Cómo no, también se pueden usar punteros con estructuras. Vamos a ver como funciona esto de los punteros con estructuras. Primero de todo hay que definir la estructura de igual forma que hacíamos antes. La diferencia está en que al declara la variable de tipo estructura debemos ponerle el operador '*****' para indicarle que es un puntero.

Creo que es importante recordar que un puntero no debe apuntar a un lugar cualquiera, debemos darle una dirección válida donde apuntar. No podemos por ejemplo crear un puntero a estructura y meter los datos directamente mediante ese puntero, no sabemos dónde apunta el puntero y los datos se almacenarían en un lugar cualquiera.

Y para comprender cómo funcionan nada mejor que un ejemplo. Este programa utiliza un puntero para acceder a la información de la estructura:

```
#include <stdio.h>

struct estructura_amigo {
        char nombre[30];
        char apellido[40];
        char telefono[10];
        int edad;
        };

struct estructura_amigo amigo = {
       "Juanjo",
```

```
       "Lopez",
       "592-0483",
       30
       };

struct estructura_amigo *p_amigo;

int main()
      {
      p_amigo = &amigo;
      printf( "%s tiene ", p_amigo->apellido );
      printf( "%i años ", p_amigo->edad );
      printf( "y su teléfono es el %s.\n" ,
p_amigo->telefono );
      }
```
Comprobado con DJGPP

Has la definición del puntero *p_amigo* vemos que todo era igual que antes. *p_amigo* es un puntero a la estructura *estructura_amigo*. Dado que es un puntero tenemos que indicarle dónde debe apuntar, en este caso vamos a hacer que apunte a la variable *amigo*:

```
      p_amigo = &amigo;
```

No debemos olvidar el operador **&** que significa 'dame la dirección donde está almacenado...'.

Ahora queremos acceder a cada campo de la estructura. Antes lo hacíamos usando el operador '.', pero, como muestra el ejemplo, si se trabaja con punteros se debe usar el operador '->'. Este operador viene a significar algo así como: "dame acceso al miembro ... del puntero ...".

Ya sólo nos queda saber cómo podemos utilizar los punteros para introducir datos en las estructuras. Lo vamos a ver un ejemplo:

```
#include <stdio.h>

struct estructura_amigo {
       char nombre[30];
       char apellido[40];
       int edad;
       };

struct estructura_amigo amigo, *p_amigo;

int main()
     {
     p_amigo = &amigo;
```

```
     /* Introducimos los datos mediante punteros
*/
     printf("Nombre: ");fflush(stdout);
     gets(p_amigo->nombre);
     printf("Apellido: ");fflush(stdout);
     gets(p_amigo->apellido);
     printf("Edad: ");fflush(stdout);
     scanf( "%i", &p_amigo->edad );

      /* Mostramos los datos */
      printf( "El amigo %s ", p_amigo->nombre );
      printf( "%s tiene ", p_amigo->apellido );
      printf( "%i años.\n", p_amigo->edad );
      }
```
Comprobado con DJGPP

NOTA: *p_amigo* es un puntero que apunta a la estructura *amigo*. Sin embargo *p_amigo->edad* es una variable de tipo int. Por eso al usar el scanf tenemos que poner el &.

[Arriba]

Punteros a arrays de estructuras

Por supuesto también podemos usar punteros con arrays de estructuras. La forma de trabajar es la misma, lo único que tenemos que hacer es asegurarnos que el puntero inicialmente apunte al primer elemento, luego saltar al siguiente hasta llegar al último.

```
#include <stdio.h>

#define ELEMENTOS       3

struct estructura_amigo {
       char nombre[30];
       char apellido[40];
       char telefono[10];
       int edad;
       };

struct estructura_amigo amigo[] =
       {
       "Juanjo", "Lopez", "504-4342", 30,
       "Marcos", "Gamindez", "405-4823", 42,
       "Ana", "Martinez", "533-5694", 20
       };

struct estructura_amigo *p_amigo;

int main()
```

```
     {
     int num_amigo;
     p_amigo = amigo;  /* apuntamos al primer
elemento del array */

     /* Ahora imprimimos sus datos */
     for( num_amigo=0; num_amigo<ELEMENTOS;
num_amigo++ )
          {
          printf( "El amigo %s ", p_amigo->nombre
);
          printf( "%s tiene ", p_amigo->apellido
);
          printf( "%i años ", p_amigo->edad );
          printf( "y su teléfono es el %s.\n" ,
p_amigo->telefono );
          /* y ahora saltamos al siguiente
elemento */
          p_amigo++;
          }
     }
```
Comprobado con DJGPP

En vez de *p_amigo = amigo;* se podía usar la forma *p_amigo = &amigo[0];*, es decir que apunte al primer elemento (el elemento 0) del array. La primera forma creo que es más usada pero la segunda quizás indica más claramente al lector principiante lo que se pretende.

Ahora veamos el ejemplo anterior de cómo introducir datos en un array de estructuras mediante punteros:

```
#include <stdio.h>

#define ELEMENTOS       3

struct estructura_amigo {
       char nombre[30];
       char apellido[40];
       int edad;
       };

struct estructura_amigo amigo[ELEMENTOS],
*p_amigo;

int main()
     {
     int num_amigo;
     /* apuntamos al primer elemento */
     p_amigo = amigo;

     /* Introducimos los datos mediante punteros
*/
     for ( num_amigo=0; num_amigo<ELEMENTOS;
num_amigo++ )
          {
```

```
          printf("Datos amigo %i\n",num_amigo);
          printf("Nombre: ");fflush(stdout);
          gets(p_amigo->nombre);
          printf("Apellido: ");fflush(stdout);
          gets(p_amigo->apellido);
          printf("Edad: ");fflush(stdout);
          scanf( "%i", &p_amigo->edad );
          /* vaciamos el buffer de entrada */
          while(getchar()!='\n');
          /* saltamos al siguiente elemento */
          p_amigo++;
          }
     /* Ahora imprimimos sus datos */
     p_amigo = amigo;
     for( num_amigo=0; num_amigo<ELEMENTOS;
num_amigo++ )
          {
          printf( "El amigo %s ", p_amigo->nombre
);
          printf( "%s tiene ", p_amigo->apellido
);
          printf( "%i años.\n", p_amigo->edad );
          p_amigo++;
          }
     }
```
Comprobado con DJGPP

Es importante no olvidar que al terminar el primer bucle for el puntero p_amigo apunta al último elemento del array de estructuras. Para mostrar los datos tenemos que hacer que vuelva a apuntar al primer elemento y por eso usamos de nuevo p_amigo=amigo; (en negrita).

[Arriba]

Paso de estructuras a funciones

Las estructuras se pueden pasar directamente a una función igual que hacíamos con las variables. Por supuesto en la definición de la función debemos indicar el tipo de argumento que usamos:

```
int nombre_función ( struct
nombre_de_la_estructura nombre_de_la
variable_estructura )
```

En el ejemplo siguiente se usa una función llamada *suma* que calcula cual será la edad 20 años más tarde (simplemente suma 20 a la edad). Esta función toma como argumento la variable estructura *arg_amigo*.

Cuando se ejecuta el programa llamamos a *suma* desde *main* y en esta variable se copia el contenido de la variable *amigo*.

Esta función devuelve un valor entero (porque está declarada como int) y el valor que devuelve (mediante return) es la suma.

```c
#include <stdio.h>

struct estructura_amigo {
        char nombre[30];
        char apellido[40];
        char telefono[10];
        int edad;
        };

int suma( struct estructura_amigo
arg_amigo )
    {
    return arg_amigo.edad+20;
    }
int main()
      {
      struct estructura_amigo amigo = {
             "Juanjo",
             "López",
             "592-0483",
             30
       };
      printf( "%s tiene ", amigo.apellido
);
      printf( "%i años ", amigo.edad );
      printf( "y dentro de 20 años tendrá
%i.\n", suma(amigo) );
      system( "PAUSE" );
      }
```

Fichero: estructuras_funciones.c

Si dentro de la función *suma* hubiésemos cambiado algún valor de la estructura, dado que es una copia no hubiera afectado a la variable *amigo* de main. Es decir, si dentro de 'suma' hacemos arg_amigo.edad = 20; el valor de arg_amigo cambiará, pero el de *amigo* seguirá siendo 30.

También se pueden pasar estructuras mediante punteros o se puede pasar simplemente un miembro (o campo) de la estructura.

Si usamos punteros para pasar estructuras como argumentos habrá que hacer unos cambios al código anterior (en negrita y subrrayado):

```c
#include <stdio.h>

...

int suma( struct estructura_amigo
*arg_amigo )
    {
    return arg_amigo->edad+20;
    }
int main()
      {
      ...
      printf( "%s tiene ", amigo.apellido
);
      printf( "%i años ", amigo.edad );
      printf( "y dentro de 20 años tendrá
%i.\n", suma(&amigo) );
      system( "PAUSE" );
      }

>
```
Fichero: estructuras_funciones2.c

Lo primero será indicar a la función suma que lo que va a recibir es un puntero, para eso ponemos el * (asterisco). Segundo, como dentro de la función suma usamos un puntero a estructura y no una variable estructura debemos cambiar el '.' (punto) por el '->'. Tercero, dentro de main cuando llamamos a *suma* debemos pasar la dirección de *amigo*, no su valor, por lo tanto debemos poner '&' delante de *amigo*.

Si usamos punteros a estructuras corremos el riesgo (o tenemos la ventaja, según cómo se mire) de poder cambiar los datos de la estructura de la variable *amigo* de main.

Pasar sólo miembros de la estructura

Otra posibilidad es no pasar toda la estructura a la función sino tan sólo los miembros que sean necesarios. Los ejemplos anteriores serían más correctos usando esta tercera opción, ya que sólo usamos el miembro 'edad':

```c
int suma( char edad )
    {
    return edad+20;
    }

int main()
      {
```

```
      printf( "%s tiene ", amigo.apellido );
      printf( "%i años ", amigo.edad );
      printf( "y dentro de 20 años tendrá %i.\n",
suma(amigo.edad) );
      }
```
Comprobado con DJGPP

Por supuesto a la función suma hay que indicarle que va a recibir una variable tipo char (amigo->edad es de tipo char).

[Arriba]

Estructuras dentro de estructuras (Anidadas)

Es posible crear estructuras que tengan como miembros otras estructuras. Esto tiene diversas utilidades, por ejemplo tener la estructura de datos más ordenada. Imaginemos la siguiente situación: una tienda de música quiere hacer un programa para el inventario de los discos, cintas y cd's que tienen. Para cada título quiere conocer las existencias en cada soporte (cinta, disco, cd), y los datos del proveedor (el que le vende ese disco). Podría pensar en una estructura así:

```
struct inventario {
        char titulo[30];
        char autor[40];
        int existencias_discos;
        int existencias_cintas;
        int existencias_cd;
        char nombre_proveedor[40];
        char telefono_proveedor[10];
        char direccion_proveedor[100];
        };
```

Sin embargo utilizando estructuras anidadas se podría hacer de esta otra forma más ordenada:

```
struct estruc_existencias {
       int discos;
       int cintas;
       int cd;
       };

struct estruc_proveedor {
       char nombre_proveedor[40];
       char telefono_proveedor[10];
       char direccion_proveedor[100];
```

```
       };

struct estruc_inventario {
        char titulo[30];
        char autor[40];
        struct estruc_existencias existencias;
        struct estruc_proveedor proveedor;
        } inventario;
```

Ahora para acceder al número de cd de cierto título usaríamos lo siguiente:

```
inventario.existencias.cd
```

y para acceder al nombre del proveedor:

```
inventario.proveedor.nombre
```

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Uniones y Enumeraciones

Contenido

Uniones

Hemos visto que las estructuras toman una parte de la memoria y se la reparten entre sus miembros. Cada miembro tiene reservado un espacio para él solo. El tamaño total que ocupa una estructura en memoria es la suma del tamaño que ocupa cada uno de sus miembros.

Las uniones tienen un aspecto similar en cuanto a cómo se definen, pero tienen una diferencia fundamental con respecto a las estructuras: los miembros comparten el mismo trozo de memoria. El espacio que ocupa en memoria una unión es el espacio que ocupa el campo más grande. Para entenderlo mejor vamos a ver un ejemplo:

Primero vamos a ver cómo se define una unión:

```
union nombre_de_la_unión
      {
      campos de la unión
      };
```

NOTA: No se debe olvidar el ';' después de cerrar llaves '}'.

Y aquí va el ejemplo:

```
union _persona
      {
      char nombre[10];
      char inicial;
      };
```

Creamos una unión y sus elementos son un *nombre* de 10 bytes (nombre[10]) y la inicial (1 byte). Como hemos dicho la unión ocupa el espacio de su elemento más grande, en este caso *nombre*. Por lo tanto la unión ocupa 10 bytes. Las variables nombre e inicial comparten el mismo sitio de la memoria. Si accedemos a *nombre* estaremos accediendo a los primeros 10 bytes de la unión (es decir, a toda la unión), si accedemos a *inicial* lo que tendremos es el primer byte de la unión.


```
#include <stdio.h>

union _persona
      {
      char nombre[10];
      char inicial;
      } pers;

int main()
     {
     printf("Escribe tu nombre: ");
     gets(pers.nombre);
     printf("\nTu nombre es: %s\n", pers.nombre);
     printf("Tu inicial es: %c\n", pers.inicial);
     }
```

Comprobado con DJGPP

Ejecutando el programa:

```
Escribe tu nombre: Gorka

Tu nombre es: Gorka
Tu inicial es: G
```

Para comprender mejor eso de que comparten el mismo espacio en memoria vamos a ampliar el ejemplo. Si añadimos unas líneas al final que modifiquen sólo la inicial e imprima el nuevo nombre:

```
#include <stdio.h>

union _persona
      {
      char nombre[10];
```

```
      char inicial;
      } pers;

int main()
     {
     printf("Escribe tu nombre: ");
     gets(pers.nombre);
     printf("\nTu nombre es: %s\n", pers.nombre);
     printf("Tu inicial es: %c\n", pers.inicial);
     /* Cambiamos la inicial */
     pers.inicial='Z';
     printf("\nAhora tu nombre es: %s\n",
pers.nombre);
     printf("y tu inicial es: %c\n",
pers.inicial);
     }
```

Comprobado con DJGPP

Tendremos el siguiente resultado:

```
Escribe tu nombre: gorka

Tu nombre es: gorka
Tu inicial es: g

Ahora tu nombre es: Zorka
y tu inicial es: Z
```

Aquí queda claro que al cambiar el valor de la inicial estamos cambiando también el nombre porque la inicial y la primera letra del nombre son la misma posición de la memoria.

Con las uniones podemos usar punteros de manera similar a lo que vimos en el capítulo de las estructuras.

[Arriba]

Enumeraciones

En el capítulo de la constantes (**cap. 4**) vimos que se podía dar nombre a las constantes con *#define*. Utilizando esta técnica podríamos hacer un programa en que definiríamos las constantes *primero...quinto*.

```
#include <stdio.h>

#define primero 1
```

```
#define segundo 2
#define tercero 3
#define cuarto  4
#define quinto  5

int main()
    {
    int posicion;
    posicion=segundo;
    printf("posicion = %i\n", posicion);
    }
```
Comprobado con DJGPP

Sin embargo existe otra forma de declarar estas constantes y es con las enumeraciones. Las enumeraciones se crean con enum:

```
enum nombre_de_la_enumeración
    {
    nombres de las constantes
    };
```

Por ejemplo:

```
enum { primero, segundo, tercero, cuarto, quinto
};
```

De esta forma hemos definido las constantes *primero, segundo,... quinto*. Si no especificamos nada la primera constante (primero) toma el valor 0, la segunda (segunda) vale 1, la tercera 2,... Podemos cambiar estos valores predeterminados por los valores que deseemos:

```
enum { primero=1, segundo, tercero, cuarto, quinto
};
```

Ahora primero vale 1, segundo vale 2, tercero vale 3,... Cada constante toma el valor de la anterior más uno. Si por ejemplo hacemos:

```
enum { primero=1, segundo, quinto=5, sexto,
septimo };
```

Tendremos: primero=1, segundo=2, quinto=5, sexto=6, septimo=7.

Con esta nueva técnica podemos volver a escribir el ejemplo de antes:

```
#include <stdio.h>

enum { primero=1, segundo, tercero, cuarto, quinto
} posicion;

int main()
    {
    posicion=segundo;
    printf("posicion = %i\n", posicion);
    }
```

Comprobado con DJGPP

Las constantes definidas con enum **sólo** pueden tomar valores enteros (pueden ser negativos). Son equivalentes a las variables de tipo *int*.

Un error habitual suele ser pensar que es posible imprimir el nombre de la constante:

```
#include <stdio.h>

enum { primero=1, segundo, tercero, cuarto, quinto
} posicion;

int main()
    {
    printf("posicion = %s\n", segundo);
    }
```

Este ejemplo contiene errores

Es habitual pensar que con este ejemplo tendremos como resultado: *posicion = segundo*. Pero no es así, en todo caso tendremos: *posicion = (null)*

Debemos pensar en las enumeraciones como una forma de hacer el programa más comprensible para los humanos, en nuestro ejemplo el ordenador donde vea *segundo* pondrá un 2 en su lugar.

[Arriba]

Ejercicios

Ejercicio :

Solución:

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Asignación dinámica de memoria

Contenido

¿Qué es la asignación dinámica?

Hasta ahora hemos visto que cada vez que queremos usar una variable debemos reservarle un lugar de la memoria al comenzar el programa. Debemos indicar cuánta memoria vamos a usar. Pero hay ocasiones en que esto no es bueno, hay veces en las que no sabemos cuánta memoria vamos a necesitar. Por ejemplo si hacemos un editor de texto no podemos saber de antemano cuál va a ser la longitud del texto.

Por eso a veces es necesario poder reservar memoria según se va necesitando. Además de esta forma nuestros programas aprovecharán mejor la memoria del ordenador en el que se ejecuten, usando sólo lo necesario.

La asignación de memoria es la base de lo que veremos en capítulos posteriores, especialmente los capítulos de listas enlazadas. En éstos capítulos es donde se verá la verdadera utilidad de la asignación dinámica.

[Arriba]

Malloc y free

Existen varias funciones para la asignación dinámica de la memoria, pero las dos funciones básicas son *malloc* y *free*.

La función malloc (**M**emory **alloc**ate - asignar memoria) reservar una parte de la memoria y devuelve la dirección del comienzo de esa parte. Esta dirección podemos almacenarla en un puntero y así podemos acceder a la memoria reservada. La función malloc tiene el siguiente formato:

```
puntero = (tipo_de_variable *) malloc( número de bytes a reservar );
```

- *puntero*: es una variable tipo puntero que almacena la dirección del bloque de memoria reservado. Puede ser un puntero a char, int, float,...
- (*tipo_de_variable* *): es lo que se llama un *molde*. La función malloc nos reserva una cierta cantidad de bytes y devuelve un puntero del tipo **void** (que es uno genérico). Con el molde le indicamos al compilador que lo convierta en un puntero del mismo tipo que la variable *puntero*. Esto no es necesario en C, ya que lo hace automáticamente, aunque es aconsejable acostumbrarse a usarlo.

Una vez reservada la memoria y guardada su dirección en un puntero podemos usar ese puntero como hemos visto hasta ahora.

Si no había suficiente memoria libre malloc devolverá el valor NULL. El puntero por tanto apuntará a NULL. Es muy importante comprobar siempre si se ha podido reservar memoria o no comprobando el valor de *puntero*:

```
if (puntero)
```

se cumple si hay memoria suficiente, en caso contrario el falso.

Cuando ya no necesitemos más el espacio reservado debemos liberarlo, es decir, indicar al ordenador que puede destinarlo a otros fines. Si no liberamos el espacio que ya no necesitamos corremos el peligro de agotar la memoria del ordenador. Para ello usamos la función *free*, que funciona de la siguiente manera:

```
free( puntero );
```

Donde *puntero* es un puntero que apunta al comienzo del bloque que habíamos reservado. Es muy importante no perder la dirección del comienzo del bloque, pues de otra forma no podremos liberarlo.

Vamos a ver un sencillo ejemplo del manejo de `malloc` y `free`:

```
#include <stdio.h>

int main()
    {
```

```
    int bytes;
    char *texto;

    printf("Cuantos bytes quieres reservar: ");
    scanf("%i",&bytes);
    texto = (char *) malloc(bytes);
    /* Comprobamos si ha tenido éxito la operación
*/
    if (texto)
       {
       printf("Memoria reservada: %i bytes = %i
kbytes = %i Mbytes\n",
              bytes, bytes/1024,
bytes/(1024*1024));
       printf("El bloque comienza en la dirección:
%p\n", texto);
       /* Ahora liberamos la memoria */
       free( texto );
       }
    else
        printf("No se ha podido reservar
memoria\n");
    }
```
Comprobado con DJGPP

En este ejemplo se pregunta cuánta memoria se quiere reservar, si se consigue se indica cuánta memoria se ha reservado y dónde comienza el bloque. Si no se consigue se indica meniante el mensaje: "No se ha podido reservar memoria".

Probemos este ejemplo reservando 1000 bytes:

```
Cuantos bytes quieres reservar: 1000
Memoria reservada: 1000 bytes = 0 kbytes = 0
Mbytes
El bloque comienza en la dirección: 90868
```

Pruébalo unas cuantas veces incrementando el tamaño de la memoria que quieres reservar. Si tienes 32Mbytes de memoria RAM teclea lo siguiente:

```
Cuantos bytes quieres reservar: 32000000
No se ha podido reservar memoria
```

Malloc no ha podido reservar tanta memoria y devuelve (null) por lo que se nos avisa de que la operación no se ha podido realizar.

[Arriba]

Ejercicios

Ejercicio :

Solución:

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Tipos de datos definidos por el usuario

Contenido

Typedef

Ya conocemos los tipos de datos que nos ofrece C: char, int, float, double con sus variantes unsigned, arrays y punteros. Además tenemos las estructuras. Pero existe una forma de dar nombre a los tipos ya establecidos y a sus posibles variaciones: usando *typedef*. Con typedef podemos crear nombres para los tipos de datos ya existentes, ya sea por capricho o para mejorar la legibilidad o la portabilidad del programa.

Por ejemplo, hay muchos programas que definen muchas variables del tipo *unsigned char*. Imaginemos un hipotético programa que dibuja una gráfica:

```
unsigned char x0, y0;      /* Coordenadas del
punto origen */
unsigned char x1, y1;      /* Coordenadas del
punto final */
unsigned char x, y;/* Coordenadas de un punto
genérico */
int F;             /* valor de la función en el
punto (x,y) */
int i;             /* Esta la usamos como contador
para los bucles */
```

La definición del tipo *unsigned char* aparte de ser larga no nos da información de para qué se usa la variable, sólo del tipo de dato que es.

Para definir nuestros propios tipos de datos debemos usar typedef de la siguiente forma:

```
typedef tipo_de_variable nombre_nuevo;
```

En nuestro ejemplo podríamos hacer:

```
typedef unsigned char COORD;
typedef int CONTADOR;
typedef int VALOR;
```

y ahora quedaría:

```
COORD x0, y0;      /* punto origen */
COORD x1, y1;      /* punto final */
COORD x, y; /* punto genérico */
VALOR F;    /* valor de la función en el punto
(x,y) */
CONTADOR i;
```

Ahora, nuestros nuevos tipos de datos, aparte de definir el tipo de variable nos dan información adicional de las variables. Sabemos que *x0*, *y0*, *x1*, *y1*, *x*, *y* son coordenadas y que *i* es un contador sin necesidad de indicarlo con un comentario.

Realmente **no estamos creando nuevos tipos de datos**, lo que estamos haciendo es **darles un nombre más cómodo para trabajar y con sentido para nosotros**.

[Arriba]

Punteros

También podemos definir tipos de punteros:

```
int *p, *q;
```

Podemos convertirlo en:

```
typedef int *ENTERO;

ENTERO p, q;
```

Aquí *p, q* son punteros aunque no lleven el operador '*', puesto que ya lo lleva *ENTERO* incorporado.

[Arriba]

Arrays

También podemos declarar tipos array. Esto puede resultar más cómodo para la gente que viene de BASIC o PASCAL, que estaban acostumbrados al tipo *string* en vez de *char*:

```
typedef char STRING[255];

STRING nombre;
```

donde nombre es realmente *char nombre[255];*.

[Arriba]

Estructuras

También podemos definir tipos de estructuras. Antes, sin typedef:

```
struct _complejo {
    double real;
    double imaginario;
    }

struct _complejo numero;
```

Con typedef:

```
typedef struct {
    double real;
    double imaginario;
    } COMPLEJO

COMPLEJO numero;
```

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Redireccionamiento

Contenido

¿Qué es la redirección?

Cuando ejecutamos el comando *dir* se nos muestra el resultado en la pantalla. Sin embargo podemos hacer que el resultado se guarde en un fichero haciendo:

```
dir > resultado.txt
```

De esta forma el listado de directorios quedará guardado en el fichero *resultado.txt* y no se mostrará en la pantalla.

Esto es lo que se llama **redirección**. En este ejemplo lo que hemos hecho ha sido redireccionar la salida utilizando '>'.

La salida del programa se dirige hacia la salida estándar (stdout, standard output). Normalmente, si no se especifica nada, la salida standard es la pantalla.

La entrada de datos al programa se hace desde la entrada estándar (stdin, standard input). Normalmente, por defecto es el teclado.

Existe una tercera opción que es la salida de error estándar (stderr, standard error). Aquí es donde se muestran los mensajes de error del programa al usuario. Normalmente suele ser la pantalla, al igual que la de salida. ¿Por qué tenemos stderr y stdout? Porque de esta forma podemos redireccionar la salida a un fichero y aún así podemos ver los mensajes de error en pantalla.

Stdin, stdout y stderr en realidad hay que verlos como ficheros:

- Si stdout es la pantalla, cuando usemos printf, el resultado se muestra en el monitor de nuestro ordenador. Podemos imaginar que stdout es un fichero cuyo contenido se muestra en la pantalla.
- Si stdin es el teclado imaginemos que lo que tecleamos va a un fichero cuyo contenido lee el programa.

[Arriba]

Redireccionar la salida

Ya hemos visto cómo se redirecciona la salida con el dir. Ahora vamos a aplicarlo a nuestro curso con un sencillo ejemplo. Vamos a ver un programa que toma los datos del teclado y los muestra en la pantalla. Si el usuario teclea 'salir' el programa termina:

```
#include <stdio.h>

int main()
    {
    char texto[100];

    gets(texto);
    do {
       printf( "%s\n",texto );
       gets(texto);
       } while ( strcmp(texto, "salir") != 0 );
    fprintf( stderr, "El usuario ha tecleado
'salir'" );
    }
```

Comprobado con DJGPP

En este programa tenemos una función nueva: **fprintf**. Esta función permite escribir en el fichero indicado. Su formato es el siguiente:

```
    int fprintf(FILE *fichero, const char
*formato, ...);
```

Ya veremos esta función más adelante. Por ahora nos basta con saber que funciona como printf pero nos permite indicar en qué fichero queremos que se escriba el texto. En nuestro ejemplo *fichero=stderr*.

Como ya hemos visto antes *stderr* es un fichero. Stderr es la pantalla, así que, si escribimos el texto en el fichero stderr lo podremos ver en el monitor.

Si ejecutamos el programa como hemos hecho hasta ahora tendremos el siguiente resultado (en negrita lo que tecleamos nosotros):

```
primera línea
primera línea
segunda
segunda
esto es la monda
esto es la monda
salir
El usuario ha tecleado 'salir'
```

Como vemos el programa repite lo que tecleamos.

Si ahora utilizamos la redirección '> **resultado.txt**' el resultado por pantalla será (en negrita lo que tecleamos nosotros):

```
primera línea
segunda
esto es la monda
salir
El usuario ha tecleado 'salir'
```

NOTA: Seguimos viendo el mensaje final (*El usuario ha tecleado 'salir'*) porque stderr sigue siendo la pantalla.

y se habrá creado un fichero llamado *resultado.txt* cuyo contenido será:

```
primera línea
segunda
esto es la monda
```

Si no hubiésemos usado stderr el mensaje final hubiese ido a stdout (y por lo tanto al fichero), así que no podríamos haberlo visto en la pantalla. Hubiera quedado en el fichero resultado.txt.

[Arriba]

Redireccionar la salida con >>

Existe una forma de redireccionar la salida de forma que se añada a un fichero en vez de sobreescribirlo. Para ello debemos usar '>>' en vez de '>'. Haz la siguiente prueba:

```
dir > resultado.txt
dir >> resultado.txt
```

Tendrás el listado del directorio dos veces en el mismo fichero. La segunda vez que llamamos al comando dir, si usamos >>, se añade al final del fichero.

Aviso

Todo esto nos sirve como una introducción al mundo de los ficheros, pero puede no ser la forma más cómoda para trabajar con ella (aunque como hemos visto es muy sencilla). El tema de los ficheros lo veremos más a fondo en el siguiente capítulo.

[Arriba]

Redireccionar la entrada

Ahora vamos a hacer algo curioso. Vamos a crear un fichero llamado *entrada.txt* y vamos a usarlo como entrada de nuestro programa. Vamos a redireccionar la entrada al fichero entrada.txt. El fichero entrada.txt debe contener lo siguiente:

```
Esto no lo he tecleado yo.
Se escribe sólo.
Qué curioso.
salir
```

Es importante la última línea 'salir' porque si no podemos tener unos resultados curiosos (en mi caso una sucesión infinita de 'Qué curioso.'.

Para cambiar la entrada utilizamos el símbolo '<'. Si nuestro programa lo hemos llamado **stdout.c** haremos:

```
# stdout < entrada.txt
```

y tendremos:

```
primera línea
segunda
esto es la monda
El usuario ha tecleado 'salir'
```

Increíble, hemos escrito todo eso sin tocar el teclado. Lo que sucede es que el programa toma los datos del fichero entrada.txt en vez del teclado. Cada vez que encuentra un salto de línea (el final de una línea) es equivalente a cuando pulsamos el 'Enter'.

Podemos incluso hacer una doble redirección: Tomaremos como entrada *entrada.txt* y como salida *resultado.txt*:

```
stdout < entrada.txt > resultado.txt
```

NOTA: Cambiando el orden también funciona:

```
stdout > resultado.txt < entrada.txt
```

[Arriba]

Redireccionar desde el programa - freopen

Existe una forma de cambiar la salida estándar desde el propio programa. Esto se puede conseguir utilizando la función **freopen**:

```
    FILE *freopen(const char *nombre_fichero,
const char *modo, FILE *fichero);
```

Esta función hace que el fichero al que apunta *fichero* se cierre y se abra pero apuntando a un nuevo fichero llamado *nombre_fichero*. Para redireccionar la salida con este método podemos hacer:

```
#include <stdio.h>

int main()
    {
    char texto[100];
```

```
    freopen( "resultado.txt","wb",stdout );
    gets(texto);
    do {
       printf( "%s\n",texto );
       gets(texto);
       } while ( strcmp(texto, "salir") != 0 );
    fprintf( stderr, "El usuario ha tecleado
'salir'" );
    }
```

En este programa la función freopen cierra el fichero *stdout*, que es la pantalla, y lo abre apuntando al fichero *resultado.txt*. Ya veremos esta función más adelante.

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Lectura de Ficheros

Contenido

Introducción

Hasta el capítulo anterior no habíamos visto ninguna forma de guardar permanentemente los datos y resultados de nuestros programas. En este capítulo vamos a verlo mediante el manejo de ficheros.

En el capítulo anterior usábamos la redirección para crear ficheros. Este es un sistema poco flexible para manejar ficheros. Ahora vamos a crear y modificar ficheros usando las funciones estándar del C.

Es importante indicar que los ficheros no son únicamente los archivos que guardamos en el disco duro, en C todos los dispositivos del ordenador se tratan como ficheros: la impresora, el teclado, la pantalla,...

[Arriba]

Lectura de un fichero

Para entrar en materia vamos a analizar un ejemplo que lee un fichero de texto y lo muestra en la pantalla:

```
#include <stdio.h>
```

```
int main()
    {
    FILE *fichero;
    char letra;

    fichero = fopen("origen.txt","r");
    if (fichero==NULL)
       {
       printf( "No se puede abrir el fichero.\n"
);
       exit( 1 );
       }
    printf( "Contenido del fichero:\n" );
    letra=getc(fichero);
    while (feof(fichero)==0)
          {
          printf( "%c",letra );
          letra=getc(fichero);
          }
    if (fclose(fichero)!=0)
       printf( "Problemas al cerrar el fichero\n"
);
    }
```

Comprobado con DJGPP

Vamos a analizar el ejemplo poco a poco:

El puntero FILE *

Todas las funciones de estrada/salida estándar usan este puntero para conseguir información sobre el fichero abierto. Este puntero no apunta al archivo sino a una estructura que contiene información sobre él.

Esta estructura incluye entre otras cosas información sobre el nombre del archivo, la dirección de la zona de memoria donde se almacena el fichero, tamaño del buffer.

Como dato curioso se adjunta la definición de la estructura FILE definida en el fichero *stdio.h*:

```
typedef struct {
  int   _cnt;
  char *_ptr;
  char *_base;
  int   _bufsiz;
  int   _flag;
  int   _file;
  char *_name_to_remove;
  int   _fillsize;
} FILE;
```

Abrir el fichero - fopen

Ahora nos toca abrir el fichero. Para ello usamos la función **fopen**. Esta función tiene el siguiente formato:

```
    FILE *fopen(const char *nombre_fichero, const
char *modo);
```

En el ejemplo usábamos:

```
    fichero = fopen("origen.txt","r");
```

El nombre de fichero se puede indicar directamente (como en el ejemplo) o usando una variable.

El fichero se puede abrir de diversas formas. Esto se especifica con el parámetro **modo**. Los modos posibles son:

r	Abre un fichero existente para lectura.
w	Crea un fichero nuevo (o borra su contenido si existe) y lo abre para escritura.
a	Abre un fichero (si no existe lo crea) para escritura. El puntero se sitúa al final del archivo, de forma que se puedan añadir datos si borrar los existentes.

Se pueden añadir una serie de modificadores siguiendo a los modos anteriores:

b	Abre el fichero en modo binario.
t	Abre el fichero en modo texto.
+	Abre el fichero para lectura y escritura.

Ejemplos de combinaciones:

- rb+ - Abre el fichero en modo binario para lectura y escritura.
- w+ - Crea (o lo borra si existe) un fichero para lectura y escritura.
- rt - Abre un archivo existente en modo texto para lectura.

Comprobar si está abierto

Una cosa muy importante después de abrir un fichero es comprobar si realmente está abierto. El sistema no es infalible y pueden producirse fallos: el fichero puede no existir, estar dañado o no tener permisos de lectura.

Si intentamos realizar operaciones sobre un puntero tipo FILE cuando no se ha conseguido abrir el fichero puede haber problemas. Por eso es importante comprobar si se ha abierto con éxito.

Si el fichero no se ha abierto el puntero *fichero* (puntero a FILE) tendrá el valor NULL, si se ha abierto con éxito tendrá un valor distinto de NULL. Por lo tanto para comprobar si ha habido errores nos fijamos en el valor del puntero:

```
    if (fichero==NULL)
       {
       printf( "No se puede abrir el fichero.\n"
);
       exit( 1 );
       }
```

Si fichero==NULL significa que no se ha podido abrir por algún error. Lo más conveniente es salir del programa. Para salir utilizamos la función exit(1), el 1 indica al sistema operativo que se han producido errores.

Lectura del fichero - getc

Ahora ya podemos empezar a leer el fichero. Para ello podemos utilizar la función **getc**, que lee los caracteres uno a uno. Se puede usar también la función fgetc (son equivalentes, la diferencia es que getc está implementada como macro). Además de estas dos existen otras funciones como fgets, fread que leen más de un carácter y que veremos más adelante.

El formato de la función getc (y de fgetc) es:

```
    int getc(FILE *fichero);
```

En este caso lo usamos como:

```
letra = getc( fichero );
```

Tomamos un carácter de *fichero*, lo almacenamos en *letra* y el puntero se coloca en el siguiente carácter.

Comprobar fin de fichero - feof

Cuando entramos en el bucle while, la lectura se realiza hasta que se encuentre el final del fichero. Para detectar el final del fichero se pueden usar dos formas:

- con la función *feof()*
- comprobando si el valor de *letra* es **EOF**.

En el ejemplo hemos usado la función feof. Esta función es de la forma:

```
    int feof(FILE *fichero);
```

Esta función comprueba si se ha llegado al final de *fichero* en cuyo caso devuelve un valor distinto de 0. Si no se ha llegado al final de fichero devuelve un cero. Por eso lo usamos del siguiente modo:

```
while ( feof(fichero)==0 )

o

while ( !feof(fichero) )
```

La segunda forma que comentaba arriba consiste en comprobar si el carácter leído es el de fin de fichero **EOF**:

```
while ( letra!=EOF )
```

Cuando trabajamos con ficheros de texto no hay ningún problema, pero si estamos manejando un fichero binario podemos encontrarnos EOF antes del fin de fichero. Por eso es mejor usar feof.

Cerrar el fichero - fclose

Una vez realizadas todas las operaciones deseadas sobre el fichero hay que cerrarlo. Es importante no olvidar este paso pues el fichero podría corromperse. Al cerrarlo se vacían los buffers y se guarda el fichero en disco. Un fichero se cierra mediante la función fclose(fichero). Si todo va bien fclose devuelve un cero, si hay problemas devuelve otro valor. Estos problemas se pueden producir si el disco está lleno, por ejemplo.

```
    if (fclose(fichero)!=0)
        printf( "Problemas al cerrar el fichero\n"
);
```

[Arriba]

Lectura de líneas - fgets

La función **fgets** es muy útil para leer líneas completas desde un fichero. El formato de esta función es:

```
    char *fgets(char *buffer, int longitud_max,
FILE *fichero);
```

Esta función lee desde el fichero hasta que encuentra un carácter '\n' o hasta que lee *longitud_max-1* caracteres y añade '\0' al final de la cadena. La cadena leída la almacena en *buffer*.

Si se encuentra EOF antes de leer ningún carácter o si se produce un error la función devuelve NULL, en caso contrario devuelve la dirección de *buffer*.

```
#include <stdio.h>

int main()
    {
    FILE *fichero;
    char texto[100];

    fichero=fopen("origen.txt","r");
    if (fichero==NULL)
       {
       printf( "No se puede abrir el fichero.\n"
);
       exit( 1 );
       }
    printf( "Contenido del fichero:\n" );
    fgets(texto,100,fichero);
    while (feof(fichero)==0)
          {
          printf( "%s",texto );
          fgets(texto,100,fichero);
          }
    if (fclose(fichero)!=0)
       printf( "Problemas al cerrar el fichero\n"
);
    }
```

Comprobado con DJGPP

[Arriba]

fread

Esta función la vamos a tratar en un tema posterior junto con la función fwrite. Estas dos funciones permiten guardar y leer cualquier tipo de dato, incluso estructuras.

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Escritura de Ficheros

Contenido

Introducción

En este capítulo vamos a completar la parte que nos faltaba en el anterior capítulo, escribir en un fichero.

Como en el capítulo anterior vamos a verlo con un ejemplo. En este ejemplo abrimos un fichero '*origen.txt*' y lo copiamos en otro fichero '*destino.txt*'. Además el fichero se muestra en pantalla. Las partes nuevas están marcadas en negrita para que se vea la diferencia:

```
#include <stdio.h>

int main()
    {
    FILE *origen, *destino;
    char letra;

    origen=fopen("origen.txt","r");
    destino=fopen("destino.txt","w");
    if (origen==NULL || destino==NULL)
       {
       printf( "Problemas con los
ficheros.\n" );
       exit( 1 );
       }
    letra=getc(origen);
    while (feof(origen)==0)
          {
          putc(letra,destino);
```

```
          printf( "%c",letra );
          letra=getc(origen);
          }
    if (fclose(origen)!=0)
       printf( "Problemas al cerrar el
fichero origen.txt\n" );
    if (fclose(destino)!=0)
       printf( "Problemas al cerrar el
fichero destino.txt\n" );
    }
```

Comprobado con DJGPP

[Arriba]

Escritura de un fichero

El puntero FILE *

Como hemos visto en el capítulo anterior el puntero FILE es la base de la escritura/lectura de archivos. Por eso definimos dos punteros FILE:

- el puntero 'origen' donde vamos a almacenar la información sobre el fichero origen.txt y
- 'destino' donde guardamos la del fichero destino.txt (el nombre del puntero no tiene por qué coincidir con el de fichero).

Abrir el fichero - fopen

El siguiente paso, como antes, es abrir el fichero usando fopen. La diferencia es que ahora tenemos que abrirlo para escritura. Usamos el modo 'w' (crea el fichero o lo vacía si existe) porque queremos crear un fichero.

Recordemos que después de abrir un fichero hay que comprobar si la operación se ha realizado con éxito. En este caso, como es un sencillo ejemplo, los he comprobado ambos a la vez:

```
if (origen==NULL || destino==NULL)
```

pero es más correcto hacerlo por separado así sabemos dónde se está produciendo el posible fallo.

Por comodidad para el lector repito aquí la lista de modos posibles:

r	Abre un fichero existente para lectura.
w	Crea un fichero nuevo (o borra su contenido si existe) y lo abre para escritura.
a	Abre un fichero (si no existe lo crea) para escritura. El puntero se sitúa al final del archivo, de forma que se puedan añadir datos si borrar los existentes.

y los modificadores eran:

b	Abre el fichero en modo binario.
t	Abre el fichero en modo texto.
+	Abre el fichero para lectura y escritura.

[Arriba]

Lectura del origen y escritura en destino- getc y putc

Como se puede observar en el ejemplo la lectura del fichero se hace igual que lo que vimos en el capítulo anterior. Para la escritura usamos la función **putc**:

```
int putc(int c, FILE *fichero);
```

donde *c* contiene el carácter que queremos escribir en el fichero y el puntero *fichero* es el fichero sobre el que trabajamos.

De esta forma vamos escribiendo en el fichero *destino.txt* el contenido del fichero *origen.txt*.

Comprobar fin de fichero

Como siempre que leemos datos de un fichero debemos comprobar si hemos llegado al final. Sólo debemos comprobar si estamos al final del fichero que leemos. No tenemos que comprobar el final del fichero en el que escribimos puesto que lo estamos creando y aún no tiene final.

[Arriba]

Cerrar el fichero - fclose

Y por fin lo que nunca debemos olvidar al trabajar con ficheros: cerrarlos. Debemos cerrar tanto los ficheros que leemos como aquellos sobre los que escribimos.

[Arriba]

Escritura de líneas - fputs

La función **fputs** trabaja junto con la función fgets que vimos en el capítulo anterior:

```
int fputs(const char *cadena, FILE
*fichero);
```

[Arriba]

Ejercicios

Ejercicio 1

Escribe un programa que lea un fichero y le suprima todas las vocales.

Es decir que siendo el fichero origen.txt:

```
El alegre campesino
pasea por el campo
ajeno a que el toro
se acerca por detrás
```

El fichero destino.txt sea:

```
l lgr cmpsn
ps pr l cmp
jn  q l tr
s crc pr dtrás
```

Ejercicio 2

La solución propuesta no elimina las vocales acentuadas, modifica el programa para conseguirlo.

Solución al ejercicio 1

```
#include <stdio.h>

int main()
    {
    FILE *origen, *destino;
    char letra;

    origen=fopen("origen.txt","r");
    destino=fopen("destino.txt","w");
    if (origen==NULL || destino==NULL)
       {
       printf( "Problemas con los
ficheros.\n" );
       exit( 1 );
       }
    letra=getc(origen);
    while (feof(origen)==0)
          {
          if (!strchr("AEIOUaeiou",letra))
putc( letra, destino );
          letra=getc(origen);
          }
    if (fclose(origen)!=0)
       printf( "Problemas al cerrar el
fichero origen.txt\n" );
    if (fclose(destino)!=0)
       printf( "Problemas al cerrar el
fichero destino.txt\n" );
    }
```

Comprobado con DJGPP

Solución al ejercicio 2

Lo único que hay que hacer es añadir "áéíóúÁÉÍÓÚ" a la línea:

```
          if
(!strchr("AEIOUaeiouáéíóúÁÉÍÓÚ",letra))
putc( letra, destino );
```

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Otras funciones para el manejo de ficheros

[Anterior] [Siguiente] [Contenido]

Contenido

Introducción

En este tema vamos a ver otras funciones que permiten el manejo de ficheros.

[Arriba]

fread y fwrite

Las funciones que hemos visto hasta ahora (getc, putc, fgets, fputs) son adecuadas para trabajar con caracteres (1 byte) y cadenas. Pero, ¿qué sucede cuando queremos trabajar con otros tipos de datos?

Supongamos que queremos almacenar variables de tipo int en un fichero. Como las funciones vistas hasta ahora sólo pueden operar con cadenas deberíamos convertir los valores a cadenas (con la función *itoa*). Para recuperar luego estos valores deberíamos leerlos como cadenas y pasarlos a enteros (*atoi*).

Existe una solución mucho más fácil. Vamos a utilizar las funciones **fread** y **fwrite**. Estas funciones nos permiten tratar con datos de cualquier tipo, incluso con estructuras.

fwrite

Empecemos con *fwrite*, que nos permite escribir en un fichero. Esta función tiene el siguiente formato:

```
size_t fwrite(void *buffer, size_t tamano,
size_t numero, FILE *pfichero);
```

- buffer - variable que contiene los datos que vamos a escribir en el fichero.
- tamano - el tamaño del tipo de dato a escribir. Puede ser un int, un float, una estructura, ... Para conocer su tamaño usamos el operador *sizeof*.
- numero - el número de datos a escribir.
- pfichero - El puntero al fichero sobre el que trabajamos.

Para que quede más claro examinemos el siguiente ejemplo: un programa de agenda que guarda el nombre, apellido y teléfono de cada persona.

```
#include <stdio.h>

struct {
        char nombre[20];
        char apellido[20];
        char telefono[15];
        } registro;

int main()
    {
    FILE *fichero;

    fichero = fopen( "nombres.txt", "a" );
    do {
       printf( "Nombre: " ); fflush(stdout);
       gets(registro.nombre);
       if (strcmp(registro.nombre,""))
          {
          printf( "Apellido: " );
fflush(stdout);
          gets(registro.apellido);
          printf( "Teléfono: " );
fflush(stdout);
          gets(registro.telefono);
          fwrite( ®istro, sizeof(registro),
1, fichero );
          }
       } while
(strcmp(registro.nombre,"")!=0);
    fclose( fichero );
    }
```

Comprobado con DJGPP

NOTA: El bucle termina cuando el 'nombre' se deja en blanco.

Este programa guarda los datos personales mediante fwrite usando la estructura *registro*. Abrimos el fichero en modo '*a*'

(append, añadir), para que los datos que introducimos se añadan al final del fichero.

Una vez abierto abrimos estramos en un bucle *do-while* mediante el cual introducimos los datos. Los datos se van almacenando en la variable *registro* (que es una estructura). Una vez tenemos todos los datos de la persona los metemos en el fichero con fwrite:

```
fwrite( ®istro, sizeof(registro), 1, fichero
);
```

- ®istro - es la variable (en este caso una estructura) que contiene la información a meter al fichero.
- sizeof(registro) - lo utillizamos para saber cuál es el número de bytes que vamos a guardar, el tamaño en bytes que ocupa la estructura.
- 1 - indica que sólo vamos a guardar un elemento. Cada vez que se recorre el bucle guardamos sólo un elemento.
- fichero - el puntero FILE al fichero donde vamos a escribir.

fread

La función *fread* se utiliza para sacar información de un fichero. Su formato es:

```
size_t fread(void *buffer, size_t tamano,
size_t numero, FILE *pfichero);
```

Siendo *buffer* la variable donde se van a escribir los datos leídos del fichero *pfichero*.

El valor que devuelve la función indica el número de elementos de tamaño 'tamano' que ha conseguido leer. Nosotros podemos pedirle a fread que lea 10 elementos (numero=10), pero si en el fichero sólo hay 6 elementos fread devolverá el número 6.

Siguiendo con el ejemplo anterior ahora vamos a leer los datos que habíamos introducido en "nombres.txt".

```
#include <stdio.h>

struct {
        char nombre[20];
        char apellido[20];
        char telefono[15];
```

```
        } registro;

int main()
    {
    FILE *fichero;

    fichero = fopen( "nombres.txt", "r" );
    while (!feof(fichero)) {
       if (fread( ®istro, sizeof(registro),
1, fichero )) {
                printf( "Nombre: %s\n",
registro.nombre );
                printf( "Apellido: %s\n",
registro.apellido);
                printf( "Teléfono: %s\n",
registro.telefono);
                }
       }
    fclose( fichero );
    }
```
Comprobado con DJGPP

Abrimos el fichero *nombres.txt* en modo lectura. Con el bucle while nos aseguramos que recorremos el fichero hasta el final (y que no nos pasamos).

La función fread lee un registro (numero=1) del tamaño de la estructura *registro*. Si realmente ha conseguido leer un registro la función devolverá un 1, en cuyo caso la condición del 'if' será verdadera y se imprimirá el registro en la pantalla. En caso de que no queden más registros en el fichero, fread devolverá 0 y no se mostrará nada en la pantalla.

[Arriba]

fseek y ftell

fseek

La función fseek nos permite situarnos en la posición que queramos de un fichero abierto. Cuando leemos un fichero hay un 'puntero' que indica en qué lugar del fichero nos encontramos. Cada vez que leemos datos del fichero este puntero se desplaza. Con la función fseek podemos situar este puntero en el lugar que deseemos.

El formato de fseek es el siguiente:

```
int fseek(FILE *pfichero, long
desplazamiento, int modo);
```

Como siempre *pfichero* es un puntero de tipo FILE que apunta al fichero con el que queremos trabajar.

desplazamiento son las posiciones (o bytes) que queremos desplazar el puntero. Este desplazamiento puede ser de tres tipos dependiendo del valor de *modo*:

SEEK_SET	El puntero se desplaza desde el principio del fichero.
SEEK_CUR	El puntero se desplaza desde la posición actual del fichero.
SEEK_END	El puntero se desplaza desde el final del fichero.

Estas tres constantes están definidas en el fichero <stdio.h>. Como curiosidad se indican a continuación sus definiciones:

```
#define SEEK_SET        0
#define SEEK_CUR        1
#define SEEK_END        2
```

Nota: es posible que los valores cambien de un compilador a otro.

Si se produce algún error al intentar posicionar el puntero, la función devuelve un valor distinto de 0. Si todo ha ido bien el valor devuleto es un 0.

En el siguiente ejemplo se muestra el funcionamiento de fseek. Se trata de un programa que lee la letra que hay en la posición que especifica el usuario.

```
#include <stdio.h>

int main()
    {
    FILE *fichero;
    long posicion;
    int resultado;

    fichero = fopen( "origen.txt", "r" );
    printf( "¿Qué posición quieres leer? "
); fflush(stdout);
    scanf( "%D", &posicion );
    resultado = fseek( fichero, posicion,
SEEK_SET );
```

```
    if (!resultado)
        printf( "En la posición %D está la
letra %c.\n", posicion, getc(fichero) );
    else
        printf( "Problemas posicionando el
cursor.\n" );
    fclose( fichero );
    }
```
Comprobado con DJGPP

ftell

Esta función es complementaria a fseek, devuelve la posición actual dentro del fichero.

Su formato es el siguiente:

```
long ftell(FILE *pfichero);
```

El valor que nos da ftell puede ser usado por fseek para volver a la posición actual.

[Arriba]

fprintf y fscanf

Estas dos funciones trabajan igual que sus equivalentes printf y scanf. La única diferencia es que podemos especificar el fichero sobre el que operar (si se desea puede ser la pantalla para fprintf o el teclado para fscanf).

Los formatos de estas dos funciones son:

```
int fprintf(FILE *pfichero, const char
*formato, ...);
```

```
int fscanf(FILE *pfichero, const char
*formato, ...);
```

[Arriba]

Ejercicios

En preparación.

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com

Listas enlazadas simples (I)

Contenido

Introducción

En el capítulo de 'Asignación dinámica de memoria' vimos que para ahorrar memoria podíamos reservarla dinámicamente (sobre la marcha). En mayor parte de los ejemplos que hemos visto hasta ahora reservábamos la memoria que íbamos a usar al comenzar el programa (al definir las variables).

El problema surge a la hora de hacer un programa al estilo de una agenda. No sabemos a priori cuántos nombres vamos a meter en la agenda, así que si usamos un array para este programa podemos quedarnos cortos o pasarnos. Si por ejemplo creamos una agenda con un array de mil elementos (que pueda contener mil números) y usamos sólo 100 estamos desperdiciando una cantidad de memoria importante. Si por el contrario decidimos crear una agenda con sólo 100 elementos para ahorrar memoria y necesitamos 200 nos vamos a quedar cortos. La mejor solución para este tipo de programas son las **listas enlazadas**.

En una lista enlazada la memoria se va tomando según se necesita. Cuando queremos añadir un nuevo elemento reservamos memoria para él y lo añadimos a la lista. Cuando queremos eliminar el elemento simplemente lo sacamos de la lista y liberamos la memoria usada.

Las listas enlazadas pueden ser simples, dobles o circulares. En este capítulo y el siguiente vamos a ver sólo las listas simples.

[Arriba]

Cómo funciona una lista

Para crear una lista necesitamos recordar nuestros conocimientos sobre estructuras y asignación dinámica de memoria. Vamos a desarrollar este tema creando una sencilla agenda que contiene el nombre y el número de teléfono.

Una lista enlazada simple necesita una estructura con varios campos, los campos que contienen los datos necesarios (nombre y teléfono) y otro campo que contiene un puntero a la propia estructura. Este puntero se usa para saber dónde está el siguiente elemento de la lista, para saber la posición en memoria del siguiente elemento.

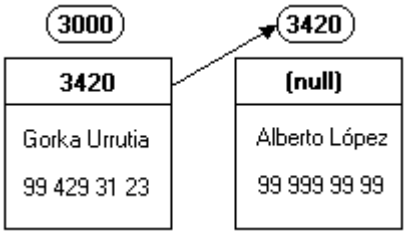
```
struct _agenda {
       char nombre[20];
       char telefono[12];
       struct _agenda *siguiente;
       };
```

Comprobado con DJGPP

Podríamos representar la estructura algo así:

Ahora supongamos que añadimos un elemento a la lista, por ejemplo mis datos: nombre="Gorka Urrutia", telefono="99 429 31 23" (el teléfono es totalmente falso :-). Lo primero que debemos hacer es reservar (con la función malloc que ya hemos visto) un espacio en memoria para almacenar el elemento. Supongamos que se almacena en la posición 3000 (por decir un número cualquiera). El puntero *siguiente* debe apuntar a NULL, ya que no hay más elementos en la lista. El elemento quedaría así:

Ahora añadimos un nuevo elemento: nombre="Alberto López" telefono="99 999 99 99". Hay que reservar (con malloc) memoria para este nuevo elemento. Vamos a imaginar que este elemento se guarda en la posición de la memoria número 3420. La lista quedaría así:



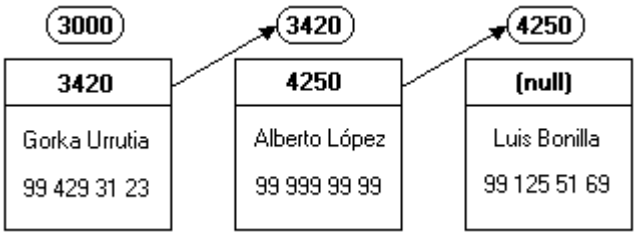
Lo primero que debemos hacer es reservar la memoria para el elemento, luego se le rellenan los datos, se pone el puntero *siguiente* apuntando a NULL (porque será el último), y decir al elemento anterior que apunte al elemento que hemos añadido.

Si quisiéramos mostrar en pantalla la lista comenzaríamos por el primer elemento, lo imprimiríamos y con el puntero siguiente saltaríamos al segundo elemento, y así hasta que el puntero *siguiente* apunte a NULL.

Siempre debemos tener un puntero del tipo _agenda para recordar la posición en memoria del primer elemento de la lista. Si perdemos este puntero perderemos la lista completa, así que mucho cuidado. Este puntero se definiría así:

```
struct _agenda primero;
```

Añadiendo otro elemento más:



[Arriba]

Ejemplo de una lista simple

Para estudiar el funcionamiento de una lista simple vamos a usar el programa agenda que hemos comentado arriba. Existen otras formas de implementar una lista simple, pero la que uso en el programa me ha parecido la más sencilla para exlicar su funcionamiento. El programa utiliza dos funciones muy importantes:

anadir_elemento
: esta función se encarga de añadir nuevos elementos a la lista.

mostrar_lista
: recorre la lista entera y muestra todos sus elementos en la pantalla.

Estas dos funciones se explican en los siguientes apartados. Echa una ojeada al código y pasa al **siguiente apartado.**

Para controlar la lista usamos dos punteros, *primero y *ultimo. Como podrás imaginar el primero guarda la posición del primer elemento y el segundo la del último.

```
 #include <stdio.h>

 struct _agenda {
        char nombre[20];
        char telefono[12];
        struct _agenda *siguiente;
        };

 struct _agenda *primero, *ultimo;

 void mostrar_menu() {
      printf("\n\nMenú:\n=====\n\n");
      printf("1.- Añadir elementos\n");
      printf("2.- Borrar elementos\n");
      printf("3.- Mostrar lista\n");
      printf("4.- Salir\n\n");
      printf("Escoge una opción:
");fflush(stdout);
 }

 /* Con esta función añadimos un elemento al
final de la lista */
 void anadir_elemento() {
      struct _agenda *nuevo;

      /* reservamos memoria para el nuevo
elemento */
      nuevo = (struct _agenda *) malloc
(sizeof(struct _agenda));
      if (nuevo==NULL) printf( "No hay
memoria disponible!\n");

      printf("\nNuevo elemento:\n");
      printf("Nombre: "); fflush(stdout);
```

```
      gets(nuevo->nombre);
      printf("Teléfono: "); fflush(stdout);
      gets(nuevo->telefono);

      /* el campo siguiente va a ser NULL
por ser el último elemento
         de la lista */
      nuevo->siguiente = NULL;

      /* ahora metemos el nuevo elemento en
la lista. lo situamos
         al final de la lista */
      /* comprobamos si la lista está vacía.
si primero==NULL es que no
         hay ningún elemento en la lista.
también vale ultimo==NULL */
      if (primero==NULL) {
         printf( "Primer elemento\n");
         primero = nuevo;
         ultimo = nuevo;
         }
      else {
           /* el que hasta ahora era el
último tiene que apuntar al nuevo */
           ultimo->siguiente = nuevo;
           /* hacemos que el nuevo sea ahora
el último */
           ultimo = nuevo;
      }
 }

 void mostrar_lista() {
      struct _agenda *auxiliar; /* lo usamos
para recorrer la lista */
      int i;

      i=0;
      auxiliar = primero;
      printf("\nMostrando la lista
completa:\n");
      while (auxiliar!=NULL) {
            printf( "Nombre: %s, Telefono:
%s\n",
                  auxiliar-
>nombre,auxiliar->telefono);
            auxiliar = auxiliar->siguiente;
            i++;
      }
      if (i==0) printf( "\nLa lista está
vacía!!\n" );
 }

 int main() {
     char opcion;

     primero = (struct _agenda *) NULL;
     ultimo = (struct _agenda *) NULL;
     do {
         mostrar_menu();
         opcion = getch();
```

```
             switch ( opcion ) {
                case '1': anadir_elemento();
                       break;
                case '2':  printf("No
disponible todavía!\n");
                        break;
                case '3':
mostrar_lista(primero);
                        break;
                case '4': exit( 1 );
                default: printf( "Opción no
válida\n" );
                         break;
             }
     } while (opcion!='4');
 }
```
Comprobado con DJGPP

[Arriba]

Añadir nuevos elementos

Reproducimos aquí de nuevo el código de la función *anadir_elemento*.

Lo primero creamos un puntero que apuntará al nuevo elemento que vamos a añadir:

```
      struct _agenda *nuevo;
```

Una vez creado el puntero tenemos que reservar un espacio en memoria donde se almacenará el nuevo elemento. Este espacio debe ser del tamaño de la estructura, que lo conocemos usando "sizeof(struct _agenda)". Hacemos que el puntero guarde la posición de ese espacio reservado.

Por supuesto comprobamos el valor del puntero para saber si la operación se ha realizado con éxito. Si no hay memoria suficiente para el puntero éste tomará el valor NULL.

```
      /* reservamos memoria para el nuevo
elemento */
      nuevo = (struct _agenda *) malloc
(sizeof(struct _agenda));
      if (nuevo==NULL) printf( "No hay
memoria disponible!\n");
```

```

```

El siguiente paso es pedir al usuario del programa que meta los datos. Estos datos se almacenarán directamente en la memoria que hemos reservado gracias al puntero que usamos.

```
      printf("\nNuevo elemento:\n");
      printf("Nombre: "); fflush(stdout);
      gets(nuevo->nombre);
      printf("Teléfono: "); fflush(stdout);
      gets(nuevo->telefono);
```

El último elemento de la lista siempre va a apuntar a NULL, de esta forma sabemos cuál es el último elemento. Dado que en este ejemplo vamos a meter los elementos siempre al final de la lista, el campo *siguiente* tiene que ser NULL.

```
      /* el campo siguiente va a ser NULL
por ser el último elemento
         de la lista */
      nuevo->siguiente = NULL;
```

El último paso es meter el elemento dentro de la lista (hasta ahora sólo teníamos un elemento aislado, que nada tenía que ver con la lista).

Antes de meterlo en la lista debemos comprobar si ya existía algún elemento antes. Para ello vamos a comprobar el valor del puntero *primero* que debería apuntar al primer elemento. Si *primero* es NULL eso significa que no hay ningún elemento en la lista, así que el nuevo elemento será a la vez el primero y el último de la lista:

```
      /* ahora metemos el nuevo elemento en
la lista. lo situamos
         al final de la lista */
      /* comprobamos si la lista está vacía.
si primero==NULL es que no
         hay ningún elemento en la lista.
también vale ultimo==NULL */
      if (primero==NULL) {
         printf( "Primer elemento\n");
         primero = nuevo;
```

```
          ultimo = nuevo;
          }
```

Si ya existía algún elemento, debemos situar el nuevo elemento después del último. Para ello hacemos que el campo *siguiente* del último elemento apunte al nuevo elemento (ultimo->siguiente = nuevo;). Una vez hecho esto hacemos que el nuevo elemento sea el último de la lista (ultimo = nuevo;).

```
      else {
           /* el que hasta ahora era el
último tiene que apuntar al nuevo */
           ultimo->siguiente = nuevo;
           /* hacemos que el nuevo sea ahora
el último */
           ultimo = nuevo;
      }
 }
```

[Arriba]

Mostrar la lista completa

Ya tenemos la forma de añadir elementos a una lista, ahora vamos a ver cómo recorrer la lista y mostrar su contenido.

Para recorrer la lista usaremos un puntero auxiliar al que en un ataque de rabiosa originalidad llamaremos *auxiliar*.

Para comenzar debemos hacer que 'auxiliar' apunte al primer elemento de la lista (auxiliar=primero). Para recorrer la lista usamos un bucle while y comprobamos el valor de 'auxiliar'. Hemos visto que el campo 'siguiente' del último elemento apuntaba a NULL, por lo tanto, cuando 'auxiliar' sea NULL sabremos que hemos llegado al final de la lista.

En cada vuelta del ciclo mostramos el elemento actual y saltamos al siguiente. El campo 'siguiente' del puntero 'auxiliar' contiene la dirección del siguiente elemento. Si hacemos que 'auxiliar' salte a la dirección almacenada en 'auxiliar->siguiente' estaremos en el siguiente elemento.

Es importante no olvidar saltar al siguiente elemento, puesto que si no lo hacemos así no habrá forma de salir del bucle (estaremos siempre en el primer elemento).

Como curiosidad se ha añadido una variable *i* con la que se cuenta el número de elementos de la lista. Si al final del bucle 'i' es cero, significa que no hay elementos en la lista.

```
 void mostrar_lista() {
      struct _agenda *auxiliar; /* lo usamos
para recorrer la lista */
      int i;

      i=0;
      auxiliar = primero;
      printf("\nMostrando la lista
completa:\n");
      while (auxiliar!=NULL) {
            printf( "Nombre: %s, Telefono:
%s\n",
                  auxiliar-
>nombre,auxiliar->telefono);
            auxiliar = auxiliar->siguiente;
            i++;
      }
      if (i==0) printf( "\nLa lista está
vacía!!\n" );
 }
```

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Las buenas costumbres

En esta sección voy a tratar de dar unas indicaciones de cual es la forma más correcta de programar. Qué aspecto dar al código fuente para que sea más legible y elegante, cómo distribuir el código fuente en distintos ficheros...

Aspecto del código

Observa el siguiente código:

```
#include <stdio.h>
int main(void){ char cadena[20];
printf("Introduce una cadena:"
);fflush(stdout);
gets(cadena);printf("has escrito:
%s",cadena);return 0;}
```

Ahora compáralo con este otro:

```
#include <stdio.h>

int main(void)
     {
     char cadena[20];
     printf( "Introduce una cadena:" );
     fflush(stdout);
     gets(cadena);
     printf("has escrito: %s",cadena);
     return 0;
     }
```

Creo que está claro cual es más conveniente. Si tienes que entregar una práctica de clase bien hecha recomendamos la segunda forma. Pero si tu código fuente es una chapuza y no sabes bien lo que estás haciendo recomendamos la primera (es una pesadilla intentar corregir un programa así). Por supuesto no me hago responsable de la posible mala leche del profesor.

Desde luego no es la única forma ni la más correcta, es simplemente mi estilo. Cada uno tiene su estilo personal, aunque conviene que tampoco sea 'muy personal' si es para compartirlo. Otra forma muy habitual es poner '{' justo después de la función:

```
#include <stdio.h>

int main(void) {
     char cadena[20];
     printf( "Introduce una cadena:" );
     fflush(stdout);
     gets(cadena);
     printf("has escrito: %s",cadena);
     return 0;
     }
```

[Arriba]

Los Comentarios

El código fuente debe estar bien documentado, de tal forma que si se lo pasamos a otra persona pueda entenderlo sin grandes dificultades. Quizá pienses: "mi código es mío y nunca se lo voy a dejar a nadie, así que para que voy a comentarlo". Parece lógico, pero imagina que lo abandonas y en el futuro quieres retocar el programa. Es muy probable que olvides por qué hiciste tal o cual cosa, (pasa muy a menudo, lo digo por experiencia) y tienes que pasarte un montón de tiempo descifrando algo que en su día entendiste perfectamente.

Aquí tienes un ejemplo de un programa mal comentado:

```
#include <stdio.h>

int main(void)  /* declaro la función main
*/
     {  /* Abro llaves */
     char cadena[20];   /* declaro una
variable llamada cadena */
     printf( "Introduce una cadena:" );  /*
Imprimo el mensaje 'Introduce una cadena */
     fflush(stdout);  /* vacío el buffer de
salida */
```

```
     gets(cadena);   /* pregunto por el
valor de cadena */
     printf("has escrito: %s",cadena);   /*
muestro el valor que han introducido */
     return 0;  /* hago que el programa
devuelva el valor 0;
     }  /* cierro llaves */
```

Estos comentarios están bien si el código forma parte de algún curso, pero no de un programa serio. Todo el mundo (que lleva un tiempo programando en C) sabe que con 'int main (void)' se declara una función y que con 'char cadena[20]' se declara una variable.

Los comentarios deben decirnos qué es lo que hace el programa, para qué sirven las variables y explicar las partes críticas o confusas del programas. Algo mas correcto sería:

```
/* Programa ejemplo creado para el Curso de
C para Principiantes */

#include <stdio.h>

int main(void)
     {
     char cadena[20];  /* En la variable
cadena se almacenará el valor a introducir
*/
     printf( "Introduce una cadena:" );
     fflush(stdout);
     gets(cadena);
     printf("has escrito: %s",cadena);
     return 0;
     }
```

Este programa es tan sencillo que no tiene puntos *oscuros* para comentar.

Es importante indicar para qué se van a usar las variables. Cuando tengamos muchas y pase el tiempo es difícil descifrar para que era cada una.

[Arriba]

Los nombres de las variables

Cuando hacemos un programa con prisas o por no pensar mucho damos a las variables cualquier nombre. Supongamos un programa como éste

```
     int i, j, k;
     char a, b, c;
     double x1, x2, x3, x4;
     char an1, ab, ac, ad, ae;
     /* ... muchas más variables */
```

Cómo podemos acordarnos al de un tiempo de qué era cada variable. Por eso es conveniente que aparte de indicar para qué sirve cada variable les demos nombres descriptivos:

```
     int num_enemigos;    /* Número de
enemigos en escanario */
     int personaje_x, personaje_y;   /*
Coordenadas del personaje */
     char balas;  /* Balas en cartuchera */
     char vida;   /* Vida restante */
     char arma;  /*modelo del arma */
     /* ... etc */
```

[Arriba]

[Anterior] [Siguiente] [Contenido]

cursoc@elrincondelc.com
http://www.elrincondelc.com/

Funciones matemáticas

Este capítulo es muy poco creativo, se trata de buscar las funciones matemáticas de la ayuda del Djgpp, traducirlas y ponerlas. Es un trabajo muy aburrido así que agradezco cualquier ayuda al respecto.

Contenido

[Arriba]

Introducción

Las funciones matemáticas se encuentran en <math.h>, algunas están en <stdlib.h>, ya se indicará en cada una. Con ellas podemos calcular raíces, exponentes, potencias, senos, cosenos, redondeos, valores absolutos y logaritmos.

Tenemos que fijarnos en el tipo de dato que hay que pasar y en el que devuelven las funciones, si es int o double.

En algunas funciones se necesitan conocimientos sobre estructuras. Así que sería mejor mirar el capítulo correspondiente antes, aunque haré una pequeña explicación.

[Arriba]

Funciones matemáticas

Aqui va la lista de las funciones ordenadas por categorías.

Trigonométricas

acos

Devuelve el arcoseno de un número.

```
#include <math.h>

double acos(double x);
```

Potencias, raíces, exponentes y logaritmos

sqrt

Devuelve la raiz cuadrada de un número.

```
#include <math.h>

double sqrt(double x);
```

Valor absoluto y redondeo

abs

Devuelve el valor absoluto de un número entero. Si el número es positivo devuelve el número tal cual, si es negativo devuelve el número cambiado de signo.

```
#include <stdlib.h>

int abs( int valor );
```

Ejemplo de uso:

```
#include <stdio.h>

void main()
    {
    int a = -30;
    int b;

    b = abs( -55 );

    printf( "Valores a = %i, abs(a) = %i, b = %i\n",
a, abs( a ), b );
    }
```

No hace falta poner la directiva *#include<stdlib.h>*.

Como podemos apreciar sólo vale para números enteros. Si queremos usar números en coma flotante (o con decimales) debemos usar **fabs**. También debemos tener cuidado y usar esta última si nuestros números int sólo llegan hasta 35000.

[Arriba]

Fe de Erratas

Fe de erratas

En este capítulo incluiré todos los terribles errores y olvidos que haya cometido a lo largo del curso. De esta manera si tenias una versión vieja del curso y hay algo que no te funciona puedes consultar aquí para ver si había algún error. Pido perdón por los errores cometidos y los que inevitablemente cometeré, pero que gracias a vuestra ayuda iré corrigiendo.

- En el capítulo de *mostrar información por pantalla*: En los ejemplos de gotoxy y clrscr olvidé incluir: #include <conio.h> *Descubierto por Juan Carlos Czerwien, corregido el 16/6/99.*
- En el capítulo de *sentencias*: Olvidé en **todos** los ejemplos poner el símbolo '**&**' en los scanf, aparte de que el texto de petición de datos debería ir en un printf. De tal forma que debería quedar:
- `printf ( "Introduce un número: " );`
- `scanf( "%i", &num );`
- En el capítulo de *Introducir datos por teclado*: Algunos ejemplos estaban incompletos, faltaba el void main() y las llaves. *Descubierto por Guillem Francès, corregido el 24/7/99.*
- En el capítulo de *sentencias, notas sobre las condiciones*: Los símbolos < y > estaban mal en algunos ejemplos. *Descubierto por Juan Carlos Czerwien, corregido el 24/8/99.*
- En este mismo capítulo algunos ejemplos se veían mal dependiendo del navegador. *Descubierto por Juan Carlos Czerwien, corregido el 24/8/99.*

cursoc@elrincondelc.com
http://www.elrincondelc.com/